PRENTICE HALL

代码整洁之道

Clean Code

A Handbook of Agile Software Craftsmanship



[美] Robert C. Martin 著

韩磊 译

细节之中自有天地，整洁成就卓越代码

人民邮电出版社
POSTS & TELECOM PRESS

目　录

版权信息

书名：代码整洁之道

ISBN：978-7-115-21687-8

- 著　　[美]Robert C. Martin

　译　　韩　磊

责任编辑　杨海玲

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

内容提要

软件质量，不但依赖于架构及项目管理，而且与代码质量紧密相关。这一点，无论是敏捷开发流派还是传统开发流派，都不得不承认。

本书提出一种观念：代码质量与其整洁度成正比。干净的代码，既在质量上较为可靠，也为后期维护、升级奠定了良好基础。作为编程领域的佼佼者，本书作者给出了一系列行之有效的整洁代码操作实践。这些实践在本书中体现为一条条规则（或称“启示”），并辅以来自现实项目的正、反两面的范例。只要遵循这些规则，就能编写出干净的代码，从而有效提升代码质量。

本书阅读对象为一切有志于改善代码质量的程序员及技术经理。书中介绍的规则均来自作者多年的实践经验，涵盖从命名到重构的多个编程方面，虽为一“家”之言，然诚有可资借鉴的价值。

关于封面

封面的图片是M104：草帽星系（The Sombrero Galaxy）。M104坐落于处女座（Virgo），距地球仅3000万光年。其核心是一个质量超大的黑洞，有100万个太阳那么重。

这幅图是否让你想起了Klingon星球（克林贡）[1]的卫星Praxis（普拉西斯）爆炸的事？我清楚地记得，在《星舰迷航 VI》中，大爆炸之后碎片四溅，飞舞出一个赤道光环的场景。至此，光环就成为科幻电影中爆炸场景的必然产物了。甚至就在《星舰迷航》系列电影的后续情节中，Alderaan（阿尔德然）的爆炸也有类似场景出现。

环绕M104的光环是什么造成的？它为何会有如此巨大的膨胀率和如此明亮而微小的内核？在我看来，仿佛那位于中心位置的黑洞勃然大怒，向星系的中心扔出了一个3万光年大的洞一般。在这场宇宙大崩塌所及范围之内的居民全都大难临头了。

超大质量的黑洞以星体为食，将星体的相当部分质量转换为能量。方程式$E = MC^2$已经足够体现杠杆作用了，但当M有一颗星体那么大的质量时，看吧！在那巨兽酒足饭饱之前，有多少星体会一头

撞进它的胃里？核心部分空洞的大小，是否说明了些什么呢？





封面图片：来自斯比泽太空望远镜

封面上的M104图片，是用来自于哈勃望远镜

的那幅著名的可见光相片（上图）和Spitzer（斯比泽）轨道探测器最新的红外影像（下图）组合而成。

在红外影像中，光环中的热粒子闪耀着穿过了中心膨胀体。这两幅影像组合起来，显现出我们从未见过的景象，展示了久远之前曾熊熊燃烧的火海。

---

[1] 系列剧《星舰迷航》（Star Trek）中的故事情节，Praxis星爆炸，由此导致联邦和Klingon达成首次和平协议。

序

乐嚼（Ga-Jol）是在丹麦最受欢迎的糖果品种之一，它浓郁的甘草味道，完美地弥补了此地潮湿且时常寒冷的天气。对于我们这些丹麦人，乐嚼的妙处还在于包装盒顶上印制的哲言慧语。今早我买了一包两件装，在其包装盒上发现这句丹麦谚语：

Ærlighed i små ting er ikke nogen lille ting.

“小处诚实非小事。”这句话正好是我想在这里说的。以小见大。本书写到了一些价值殊胜的小主题。

神在细节之中，建筑师Ludwig mies van der Rohe（路德维希·密斯·范·德·罗）[1] 如是说。这句话引发了有关软件开发、特别是敏捷软件开发中架构所处地位的若干争论。鲍勃（Bob）[2] 和我时常发现自己沉湎于此类对话中。没错，Ludwig mies van der Rohe的确专注于效用和基于宏伟架构之上的永恒建筑形式。然而，他也为自己设计的每所房屋挑选每个门把手。为什么？因为小处见大。

就TDD [3] 话题展开目前仍在继续的“辩论”时，鲍勃和我认识到，我们均同意软件架构在

开发中占据重要地位，但就其确切意义而言，我们之间还有分歧。然而，这种矛与盾孰利的讨论相对而言并不重要，因为在项目开始之时，我们理所当然应该让专业人士投入些许时间去思考及规划。20世纪90年代末期有关仅以 测试和代码驱动设计的概念已一去不返。相对于任何宏伟愿景，对细节的关注甚至是更为关键的专业性基础。首先，开发者通过小型实践获得可用于大型实践的技能和信用度。其次，宏大建筑中最细小的部分，比如关不紧的门、有点儿没铺平的地板，甚至是凌乱的桌面，都会将整个大局的魅力毁灭殆尽。这就是整洁代码之所系。

架构只是软件开发用到的借喻之一，主要用在那种等同于建筑师交付毛坯房一般交付初始软件产品 的场合。在Scrum和敏捷（Agile）的日子里，人们关注的是快速将产品推向市场。我们要求工厂全速运转、生产软件。这就是人类工厂：懂思考、会感受的编码人，他们由产品备忘或用户故事开始创造产品 。来自制造业的借喻在这种场合大行其道。例如，Scrum就从装配线式的日本汽车生产方式中获益良多。

即便是在汽车工业里，大量工作也并不在于生产而在于维护——或避免维护。对于软件而言，百分之八十或更多的工作量集中在我们美其名曰“维

护”的事情上：其实就是修修补补。与其接受西方关于制造好软件的传统看法，不如将其看作建筑工业中的房屋修理工，或者汽车领域的汽修工。日本式管理对于这种事 怎么说的呢？

大约在1951年，一种名为“全员生产维护”（Total Productive Maintenance，TPM）的质量保证手段在日本出现。它关注维护甚于关注生产。TPM的主要支柱之一是所谓的5S原则体系。5S是一套规程，用“规程”这个词，是为了读者便于理解。5S原则其实是精益（Lean）——西方视野中的一个时髦词，也是在软件领域渐领风骚的时髦词——的基石所在。正如鲍勃大叔（Uncle Bob）在前言中写到的，良好的软件实践遵循这些规程：专注、镇定和思考。这并非总只有关实作，有关推动工厂设备以最高速度运转。5S哲学包括以下概念：

- 整理（Seiri）[4]，或谓组织（想想英语中的sort（分类、排序）一词）。搞清楚事物之所在——通过恰当地命名之类的手段——至关重要。觉得命名标识无关紧要？读读后面的章节吧。
- 整顿（Seiton），或谓整齐（想想英文中的systematize（系统化）一词）。有句美国老话说：物皆有其位，而后物尽归其位（A place for everything, and everything in its place）。每段

代码都该在你希望它所在的地方——如果不在那里，就需要重构了。

- 清楚（Seiso），或谓清洁（想想英文中的shine（锃亮）一词）。清理工作地的拉线、油污和边角废料。对于那种四处遗弃的带注释的代码及反映过往或期望的无注释代码，本书作者怎么说的来着？除之而后快。
- 清洁（Seiketsu），或谓标准化。有关如何保持工作地清洁的组内共识。本书有没有提到在开发组内使用一贯的代码风格和实践手段？这些标准从哪里来？读读看。
- 身美（Shitsuke）[5]，或谓纪律（自律）。在实践中贯彻规程，并时时体现于个人工作上，而且要乐于改进。

如果你接受挑战——没错，就是挑战，阅读并应用本书，你就会理解和赞赏上述最后一条。我们最终是在驶向一种负责任的专业精神之根源所在，这种专业性隶属于一个关注产品生命周期的专业领域。在我们遵循TPM来维护机动车和其他机械时，停机维护——等待缺陷显现出来——并不常见。我们更上一层楼：每天检查机械，在磨损机件停止工作之前就换掉它，或者按常例每1000英里（约1609.3km）就更换润滑油、防止磨损和开裂。对于代码，应无情地做重构。还可以更进一步，就像TPM运动在50多年前的创新：一开始就打造更易维

护的机械。写出可读的代码，重要程度不亚于写出可执行的代码。1960年左右，围绕TPM引入的终极实践（ultimate practice），关注用全新机械替代旧机械。诚如Fred Brooks所言，我们或许应该每7年就重做一次软件的主要模块，清理缓慢陈腐的代码。也许我们该把重构周期从以年计缩短到以周、以天甚至以小时计。那便是细节所在了。

细节中自有天地，而在生活中应用此类手段时也有微言大义，就像我们一成不变地对那些源自日本的做法寄予厚望一般。这并非只是东方的生活观；英美民间也遍是这类警句。上引“整顿”（Seiton）二字就曾出现在某位俄亥俄州牧师的笔下，他把齐整看作是“荡涤种种罪恶之良方”。“清楚”（Seiso）又如何呢？整洁近乎虔诚（Cleanliness is next to godliness）。一张脏乱的桌子足以夺去一所丽宅的光彩。老话怎么说“身美”（Shitsuke）的？守小节者不亏大节 （He who is faithful in little is faithful in much）。对于时时准备在恰当时机做重构，为未来的“大”决定夯实基础，而不是置诸脑后，有什么说法吗？及时一针省九针 （A stitch in time saves nine）。早起的鸟儿有虫吃 （The early bird catches the worm）。日事日毕（Don’t put off until tomorrow what you can do today）。在精益实践落入软件咨询师之手前，这就是其所谓“最后时机”的本义所在。摆正单项工作在

整体中的位置呢？巨木生于树籽 （Mighty oaks from little acorns grow）。如何在日常生活中做好简单的防备性工作呢？防病好过治病 （An ounce of prevention is worth a pound of cure）。一天一苹果，医生远离我 （An apple a day keeps the doctor away）。整洁代码以其对细节的关注，荣耀了深埋于我们现有、或曾有、或该有的壮丽文化之下的智慧根源。

即便是在宏伟的建筑作品中，我们也听到关注细节的回响。想想Ludwig mies van der Rohe的门把手吧。那正是整理 （seiri）。认真对待每个变量名。你当用为自己第一个孩子命名般的谨慎来给变量命名。

正如每位房主所知，此类照料和修葺永无休止。建筑师Christopher Alexander——模式与模式语言之父——把每个设计动作看作是较小的局部修复动作。他认为，设计良好结构才是建筑师的本职所在，而更大的建筑形态则当留给模式及居住者搬进的家私来完成。设计始终在持续进行，不只是在新建一个房间时，也在我们重新粉刷墙面、更换旧地毯或者换厨房水槽时。大多数艺术门类也持类似主张。在寻找其他推崇细节的人时，我们发现，19世纪法国作家Gustav Flaubert（古斯塔夫·福楼拜）名列其中。法国诗人Paul Valery（保尔 · 瓦雷里）认

为，每首诗歌都无写完之时，得持续重写，直至放弃为止。全心倾注于细节，屡见于追求卓越的行为之中。虽然这无甚新意，但阅读本书对读者仍是一种挑战，你要重拾久已弃置脑后的良好规则，自发自主，“响应改变”。

不幸的是，我们往往见不到人们把对细节的关注当作编程艺术的基础要件。我们过早地放弃了在代码上的工作，并不是因为它业已完成，而是因为我们的价值体系关注外在表现甚于关注要交付之物的本质。疏忽最终结出了恶果：坏东西一再出现。无论是在行业里还是学术领域，研究者都很重视代码的整洁问题。供职于贝尔软件生产研究实验室（Bell Labs Software Production Research）——没错，就是生产！——时，我们有些不太严密的发现，认为前后一致的缩进风格明显标志了较低的缺陷率。我们原指望将质量归因于架构、编程语言或者其他高级概念；我们的专业能力归功于对工具的掌握和各种高高在上的设计方法，至于那些安置于厂区的机器，那些编码者，他们居然通过简单地保持一致缩进风格创造了价值，这简直是一种侮辱。我在17年前就在书中写过，这种风格远不止是一种单纯的能力那么简单。日本式的世界观深知日常工作者的价值，而且，还深知工作者简单的日常行为所锻造的开发系统的价值。质量是上百万次全心投入的结果——而非仅归功于任何来自天堂的伟大方

法。这些行为简单却不简陋，也不意味着简易。相反，它们是人力所能达的不仅伟大而且美丽的造物。忽略它们，就不成其为完整的人。

当然，我仍然提倡放宽思路，也推崇根植于深厚领域知识和软件可用性的各种架构手法的价值。但本书与此无关——至少，没有明显关系。本书精妙之处，其意义之深远，不该无人赏识。它正与Peter Sommerlad、Kevlin Henny及Giovanni Asproni等真正写代码的人现今所持的观念相吻合。他们鼓吹“代码即设计”和“简单代码”。我们要谨记，界面就是程序，而且其结构也极大地反映出程序结构，但也理应始终谦逊地承认设计存在于代码中，这至关紧要。制造上的返工导致成本上升，但重做设计却创造出价值。我们应当视代码为设计——作为过程而非终点的设计——这种高尚行为的漂亮体现。耦合与内聚的架构韵律在代码中脉动。Larry Constantine以代码的形式——而不是用UML那种高高在上的抽象概念——来描述耦合与内聚。Richard Garbriel在“Abstraction Descant”（抽象刍议）一文中告诉我们，抽象即恶。代码除恶，而整洁的代码则大抵是圣洁的。

回到我那个小小的乐嚼包装盒，我想要重点提一下，那句丹麦谚语不只是教我们重视小处，更教我们小处要诚实 。这意味着对代码诚实、对同僚

坦承代码现状，最重要的是在代码问题上不自欺。是否已尽全力“把露营地清理得比来时还干净”？签入代码前是否已做重构？这可不是皮毛小事，它正高卧于敏捷价值的正中位置。Scrum有一种建议的实践，主张重构是“完成”（Done）概念的一部分。无论是架构还是代码都不强求完美，只求竭诚尽力而已。人孰无过，神亦容之（To err is human; to forgive, divine）。在Scrum中，我们使一切可见。我们晾出脏衣服。我们坦承代码状态，因为它永不完美。我们日渐成为完整的人，配得起神的眷顾，也越来越接近细节中的伟大之处。

在自己的专业领域中，我们亟需能得到的一切帮助。假使干净的地板能减少事故发生，假使归置到位的工具能提升生产力，我也会倾力做到。至于本书，在我看过的有关将精益原则应用于软件的印刷品中，是最具实用性的。那班求索者多年来并肩奋斗，不但是为求一己之进步，更将他们的知识通过和你手上正在做的事一般的工作贡献给这个行业。看过鲍勃大叔寄来的原稿之后，我发现，世界竟略有改善了。

对高瞻远瞩的练习业已结束，我要去清理自己的书桌了。

James O. Coplien于丹麦默尔鲁普

[1]　译注：20世纪中期著名现代建筑大师，秉承“少即是多”的建筑设计哲学，缔造了玻璃幕墙等现代建筑结构。

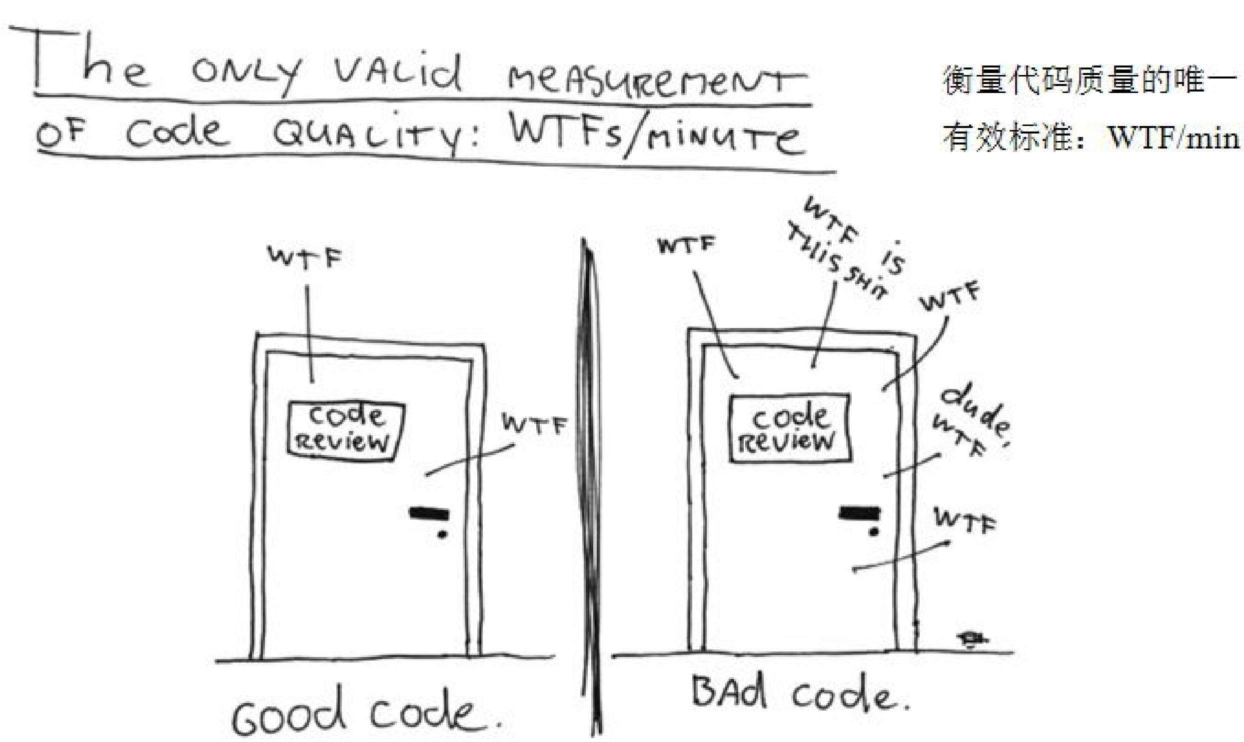
[2]　译注：本书主要作者Robert C. Martin绰号Uncle Bob，这里的“鲍勃”及后文的“鲍勃大叔”就是指Robert C. Martin。

[3]　译注：Test Driven Development，测试驱动开发。

[4]　译注：这些概念最初出现于日本，5个概念的日文罗马字拼音首字母正好都是S，所以这里也保留了日文罗马字拼音写法。中译本以日文汉字直接译出，读者留意，不可直接对应其中文意思。

[5]　译注：中文意为“素养、教养”。

前言



承Thom Holwerda惠允，自
http://www.osnews.com/story/19266/WTFs_m 再制

你的代码在哪道门后面？你的团队或公司在哪道门后面？为什么会在那里？只是一次普通的代码复查，还是产品面世后才发现一连串严重问题？我们是否在战战兢兢地调试自己之前错以为没问题的代码？客户是否在流失？经理们是否把我们盯得如芒刺在背？当事态变得严重起来，如何保证我们在那道正确的门后做补救工作？答案是：技艺（craftsmanship）。

习艺之要有二：知和行。你应当习得有关原则、模式和实践的知识，穷尽应知之事，并且要对其了如指掌，通过刻苦实践掌握它。

我可以教你骑自行车的物理学原理。实际上，经典数学的表达方式相对而言确实简洁明了。重力、摩擦力、角动量、质心等，用一页写满方程式的纸就能说明白。有了这些方程式，我可以为你证明出骑车完全可行，而且还可以告诉你骑车所需的全部知识。即便如此，你在初次骑车时还是会跌倒在地。

编码亦同此理。我们可以写下整洁代码的所有“感觉良好”的原则，放手让你去干（换言之，让你从自行车上摔下来）。那样的话，我们算是哪门子老师？而你又会成为怎样的学生呢？

不！本书可不会这么做。

学写整洁代码很难。它可不止于要求你掌握原则和模式。你得在这上面花工夫。你须自行实践，且体验自己的失败。你须观察他人的实践与失败。你须看看别人是怎样蹒跚学步，再转头研究他们的路数。你须看看别人是如何绞尽脑汁做出决策，又是如何为错误决策付出代价。

阅读本书要多用心思。这可不是那种降落前就能读完的“感觉不错”的飞机书。本书要让你用功，而且是非常用功。如何用功？阅读代码——大量代码。而且你要去琢磨某段代码好在什么地方、坏在什么地方。在我们分解，而后组合模块时，你得亦步亦趋地跟上。这得花些工夫，不过值得一试。

本书大致可分为3个部分。前几章介绍编写整洁代码的原则、模式和实践。这部分有相当多的示例代码，读起来颇具挑战性。读完这几章，就为阅读第2部分做好了准备。如果你就此止步，只能祝你好运啦！

第2部分最需要花工夫。这部分包括几个复杂性不断增加的案例研究。每个案例都清理一些代码——把有问题的代码转化为问题少一些的代码。这部分极为详细。你的思维要在讲解和代码段之间跳来跳去。你得分析和理解那些代码，琢磨每次修改的来龙去脉。

你付出的劳动将在第3部分得到回报。这部分只有一章，列出从上述案例研究中得到的启示和灵感。在遍览和清理案例中的代码时，我们把每个操作理由记录为一种启示或灵感。我们尝试去理解自己对阅读和修改代码的反应，尽力了解为什么会有这样的感受、为什么会如此行事。结果得到了一套

描述在编写、阅读、清理代码时思维方式的知识库。

如果你在阅读第2部分的案例研究时没有好好用功，那么这套知识库对你来说可能所值无几。在这些案例研究中，每次修改都仔细注明了相关启示的标号。这些标号用方括号标出，如：[H22]。由此你可以看到这些启示在何种环境下被应用和编写。启示本身不值钱，启示与案例研究中清理代码的具体决策之间的关系才有价值。

如果你跳过案例研究部分，只阅读了第1部分和第3部分，那就不过是又看了一本关于写出好软件的“感觉不错”的书。但如果你肯花时间琢磨那些案例，亦步亦趋——站在作者的角度，迫使自己以作者的思维路径考虑问题，就能更深刻地理解这些原则、模式、实践和启示。这样的话，就像一个熟练地掌握了骑车的技术后，自行车就如同其身体的延伸部分那样；对你来说，本书所介绍的整洁代码的原则、模式、实践和启示就成为了本身具有的技艺，而不再是“感觉不错”的知识。

致谢

插图

感谢两位艺术家Jennifer Kohnke和Angela Brooks。Jennifer绘制了每章起始处创意新颖、效果惊人的插图，以及Kent Beck、Ward Cunningham、Bjarne Stroustrup、Ron Jeffries、Grady Booch、Dave Thomas、Michael Feathers和我本人的肖像。

Angela绘制了文中那些精致的插图。这些年她为我画了一些画，包括*Agile Software Development: Principles, Patterns, and Practices*（中译版《敏捷软件开发：原则、模式与实践》）一书中的大量插图。她是我的长女，常给我带来极大的愉悦。

第1章　整洁代码



阅读本书有两种原因：第一，你是个程序员；第二，你想成为更好的程序员。很好。我们需要更好的程序员。

这是本有关编写好程序的书。它充斥着代码。

我们要从各个方向来考察这些代码。从顶向下，从底往上，从里而外。读完后，就能知道许多关于代码的事了。而且，我们还能说出好代码和糟糕的代码之间的差异。我们将了解到如何写出好代码。我们也会知道，如何将糟糕的代码改成好代码。

1.1　要有代码

有人也许会以为，关于代码的书有点儿落后于时代——代码不再是问题；我们应当关注模型和需求。确实，有人说过我们正在临近代码的终结点。很快，代码就会自动产生出来，不需要再人工编写。程序员完全没用了，因为商务人士可以从规约直接生成程序。

扯淡！我们永远抛不掉代码，因为代码呈现了需求的细节。在某些层面上，这些细节无法被忽略或抽象，必须明确之。将需求明确到机器可以执行的细节程度，就是编程 要做的事。而这种规约正是代码 。

我期望语言的抽象程度继续提升。我也期望领域特定语言的数量继续增加。那会是好事一桩。但那终结不了代码。实际上，在较高层次上用领域特定语言撰写的规约也将是 代码！它也得严谨、精确、规范和详细，好让机器理解和执行。

那帮以为代码终将消失的伙计，就像是巴望着发现一种无规范数学的数学家们一般。他们巴望着，总有一天能创造出某种机器，我们只要想想、嘴都不用张就能叫它依计行事。那机器要能透彻理

解我们，只有这样，它才能把含糊不清的需求翻译为可完美执行的程序，精确满足需求。

这种事永远不会发生。即便是人类，倾其全部的直觉和创造力，也造不出满足客户模糊感觉的成功系统来。如果说需求规约原则教给了我们什么，那就是归置良好的需求就像代码一样正式，也能作为代码的可执行测试来使用。

记住，代码确然是我们最终用来表达需求的那种语言。我们可以创造各种与需求接近的语言。我们可以创造帮助把需求解析和汇整为正式结构的各种工具。然而，我们永远无法抛弃必要的精确性——所以代码永存。

1.2 糟糕的代码

最近我在读Kent Beck著*Implementation Patterns*（中译版《实现模式》）[1]一书的序言。他这样写道："……本书基于一种不太牢靠的前提：好代码的确重要……"这前提不牢靠？我反对！我认为这是该领域最强固、最受支持、最被强调的前提了（我想Kent也知道）。我们知道好代码重要，是因为其短缺实在困扰了我们太久。



20世纪80年代末，有家公司写了个很流行的杀手 应用，许多专业人士都买来用。然后，发布周期开始拉长。缺陷总是不能修复。装载时间越来越久，崩溃的几率也越来越大。至今我还记得自己在某天沮丧地关掉那个程序，从此再不用它。在那之后不久，该公司就关门大吉了。

20年后，我见到那家公司的一位早期雇员，问他当年发生了什么事。他的回答叫我愈发恐惧起

来。原来，当时他们赶着推出产品，代码写得乱七八糟。特性越加越多，代码也越来越烂，最后再也没法管理这些代码了。是糟糕的代码毁了这家公司。

你是否曾为糟糕的代码所深深困扰？如果你是位有点儿经验的程序员，定然多次遇到过这类困境。我们有专用来形容这事的词：沼泽（wading）。我们趟过代码的水域。我们穿过灌木密布、瀑布暗藏的沼泽地。我们拼命想找到出路，期望有点什么线索能启发我们到底发生了什么事；但目光所及，只是越来越多死气沉沉的代码。

你当然曾为糟糕的代码所困扰过。那么——为什么要写糟糕的代码呢？

是想快点完成吗？是要赶时间吗？有可能。或许你觉得自己要干好所需的时间不够；假使花时间清理代码，老板就会大发雷霆。或许你只是不耐烦再搞这套程序，期望早点结束。或许你看了看自己承诺要做的其他事，意识到得赶紧弄完手上的东西，好接着做下一件工作。这种事我们都干过。

我们都曾经瞟一眼自己亲手造成的混乱，决定弃之而不顾，走向新一天。我们都曾经看到自己的烂程序居然能运行，然后断言能运行的烂程序总比

什么都没有强。我们都曾经说过有朝一日再回头清理。当然，在那些日子里，我们都没听过勒布朗（LeBlanc）法则：稍后等于永不（Later equals never）。

1.3　混乱的代价

只要你干过两三年编程，就有可能曾被某人的糟糕的代码绊倒过。如果你编程不止两三年，也有可能被这种代码拖过后腿。进度延缓的程度会很严重。有些团队在项目初期进展迅速，但有那么一两年的时间却慢如蜗行。对代码的每次修改都影响到其他两三处代码。修改无小事。每次添加或修改代码，都得对那堆扭纹柴了然于心，这样才能往上扔更多的扭纹柴。这团乱麻越来越大，再也无法理清，最后束手无策。

随着混乱的增加，团队生产力也持续下降，趋向于零。当生产力下降时，管理层就只有一件事可做了：增加更多人手到项目中，期望提升生产力。可是新人并不熟悉系统的设计。他们搞不清楚什么样的修改符合设计意图，什么样的修改违背设计意图。而且，他们以及团队中的其他人都背负着提升生产力的可怕压力。于是，他们制造更多的混乱，驱动生产力向零那端不断下降。如图1-1所示。

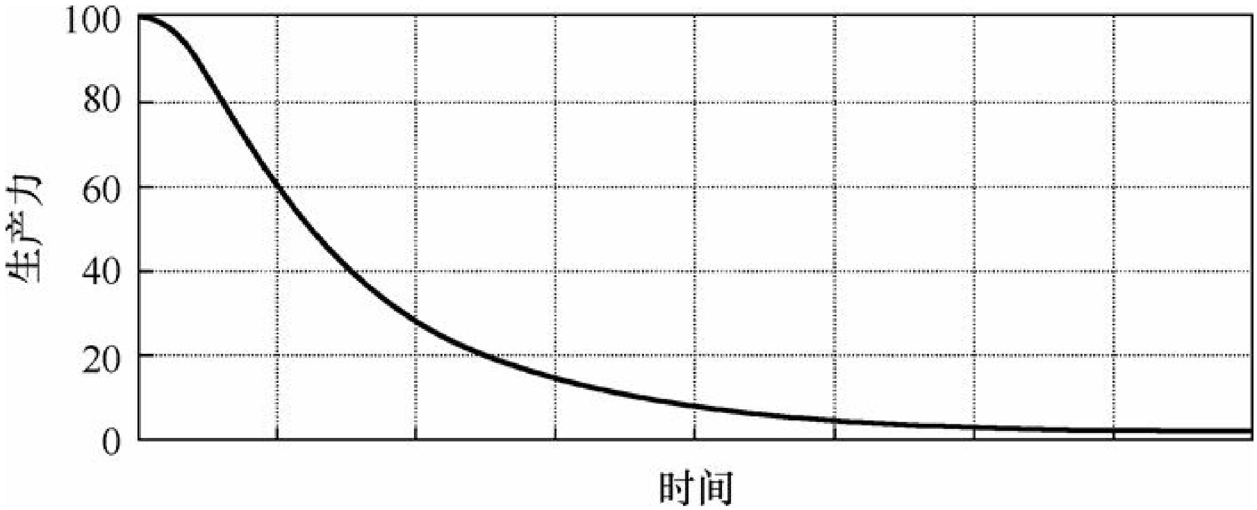


图1-1　生产力vs.时间

1.3.1　华丽新设计

最后，开发团队造反了，他们告诉管理层，再也无法在这令人生厌的代码基础上做开发。他们要求做全新的设计。管理层不愿意投入资源完全重启炉灶，但他们也不能否认生产力低得可怕。他们只好同意开发者的要求，授权去做一套看上去很美的华丽新设计。

于是就组建了一支新军。谁都想加入这个团队，因为它是张白纸。他们可以重新来过，搞出点真正漂亮的东西来。但只有最优秀、最聪明的家伙被选中。其余人等则继续维护现有系统。

现在有两支队伍在竞赛了。新团队必须搭建一套新系统，要能实现旧系统的所有功能。另外，还

得跟上对旧系统的持续改动。在新系统功能足以抗衡旧系统之前，管理层不会替换掉旧系统。

竞赛可能会持续极长时间。我就见过延续了十年之久的。到了完成的时候，新团队的老成员早已不知去向，而现有成员则要求重新设计一套新系统，因为这套系统太烂了。

假使你经历过哪怕是一小段我谈到的这种事，那么你一定知道，花时间保持代码整洁不但有关效率，还有关生存。

1.3.2　态度

你是否遇到过某种严重到要花数个星期来做本来只需数小时即可完成的事的混乱状况？你是否见过本来只需做一行修改，结果却涉及上百个模块的情况？这种事太常见了。

怎么会发生这种事？为什么好代码会这么快就变质成糟糕的代码？理由多得很。我们抱怨需求变化背离了初期设计。我们哀叹进度太紧张，没法干好活。我们把问题归咎于那些愚蠢的经理、苛求的用户、没用的营销方式和那些电话消毒剂。不过，亲爱的呆伯特（Dilbert）[2]，我们是自作自受[3]。我们太不专业了。

这话可不太中听。怎么会是自作自受呢？难道不关需求的事？难道不关进度的事？难道不关那些蠢经理和没用的营销手段的事？难道他们就不该负点责吗？

不。经理和营销人员指望从我们这里得到必须的信息，然后才能做出承诺和保证；即便他们没开口问，我们也不该羞于告知自己的想法。用户指望我们验证需求是否都在系统中实现了。项目经理指望我们遵守进度。我们与项目的规划脱不了干系，对失败负有极大的责任；特别是当失败与糟糕的代码有关时尤为如此！

“且慢！”你说。“不听经理的，我就会被炒鱿鱼。”多半不会。多数经理想要知道实情，即便他们看起来不喜欢实情。多数经理想要好代码，即便他们总是痴缠于进度。他们会奋力卫护进度和需求；那是他们该干的。你则当以同等的热情卫护代码。

再说明白些，假使你是位医生，病人请求你在给他做手术前别洗手，因为那会花太多时间，你会照办吗[4]？本该是病人说了算；但医生却绝对应该拒绝遵从。为什么？因为医生比病人更了解疾病和感染的风险。医生如果按病人说的办，就是一种不专业的态度（更别说是犯罪了）。

同理，程序员遵从不了解混乱风险的经理的意愿，也是不专业的做法。

1.3.3　谜题

程序员面临着一种基础价值谜题。有那么几年经验的开发者都知道，之前的混乱拖了自己的后腿。但开发者们背负期限的压力，只好制造混乱。简言之，他们没花时间让自己做得更快！

真正的专业人士明白，这道谜题的第二部分说错了。制造混乱无助于 赶上期限。混乱只会立刻拖慢你，叫你错过期限。赶上期限的唯一方法——做得快的唯一方法 ——就是始终尽可能保持代码整洁。

1.3.4　整洁代码的艺术

假设你相信混乱的代码是祸首，假设你接受做得快的唯一方法是保持代码整洁的说法，你一定会自问：“我怎么才能写出整洁的代码？”不过，如果你不明白整洁对代码有何意义，尝试去写整洁代码就毫无所益！

坏消息是写整洁代码很像是绘画。多数人都知道一幅画是好还是坏。但能分辨优劣并不表示懂得

绘画。能分辨整洁代码和肮脏代码，也不意味着会写整洁代码！

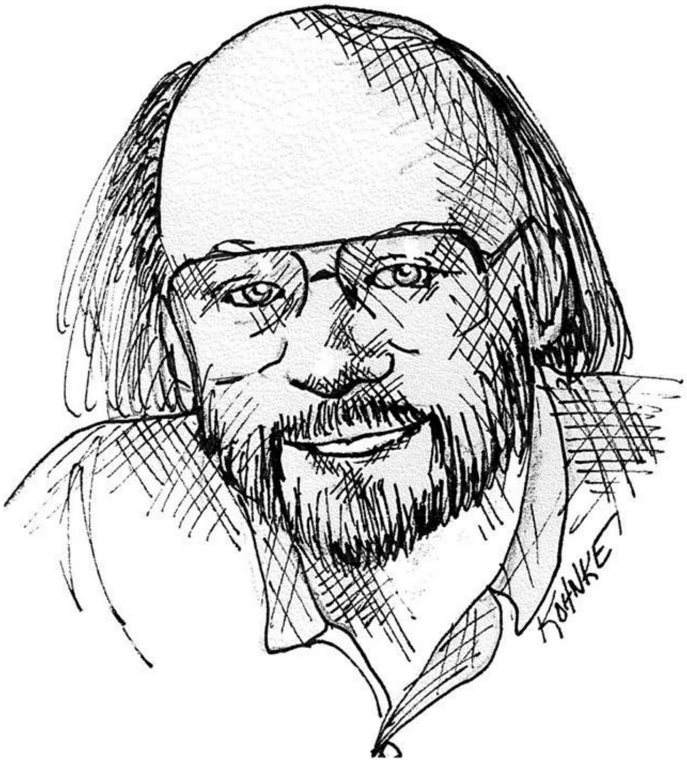
写整洁代码，需要遵循大量的小技巧，贯彻刻苦习得的“整洁感”。这种“代码感”就是关键所在。有些人生而有之。有些人费点劲才能得到。它不仅让我们看到代码的优劣，还予我们以借戒规之力化劣为优的攻略。

缺乏“代码感”的程序员，看混乱是混乱，无处着手。有“代码感”的程序员能从混乱中看出其他的可能与变化。“代码感”帮助程序员选出最好的方案，并指导程序员制订修改行动计划，按图索骥。

简言之，编写整洁代码的程序员就像是艺术家，他能用一系列变换把一块白板变作由优雅代码构成的系统。

1.3.5 什么是整洁代码

有多少程序员，就有多少定义。所以我只询问了一些非常知名且经验丰富的程序员。



Bjarne Stroustrup，**C++**语言发明者，***C++ Programming Language***（中译版《**C++**程序设计语言》）一书作者。

我喜欢优雅和高效的代码。代码逻辑应当直截了当，叫缺陷难以隐藏；尽量减少依赖关系，使之便于维护；依据某种分层战略完善错误处理代码；性能调至最优，省得引诱别人做没规矩的优化，搞出一堆混乱来。整洁的代码只做好一件事。

Bjarne用了“优雅”一词。说得好！我MacBook上的词典提供了如下定义：外表或举止上令人愉悦的优美和雅观；令人愉悦的精致和简单。 注意

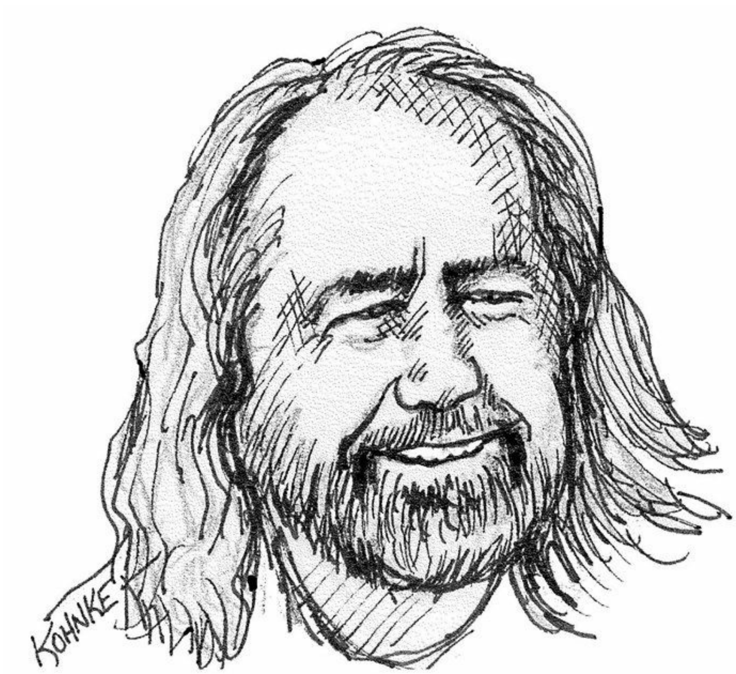
对“愉悦”一词的强调。Bjarne显然认为整洁的代码读起来令人愉悦 。读这种代码，就像见到手工精美的音乐盒或者设计精良的汽车一般，让你会心一笑。

Bjarne也提到效率——而且两次 提及。这话出自C++发明者之口，或许并不出奇；不过我认为并非是在单纯追求速度。被浪费掉的运算周期并不雅观，并不令人愉悦。留意Bjarne怎么描述那种不雅观的结果。他用了“引诱”这个词。诚哉斯言。糟糕的代码引发 混乱！别人修改糟糕的代码时，往往会越改越烂。

务实的Dave Thomas和Andy Hunt从另一角度阐述了这种情况。他们提到破窗理论 [5] 。窗户破损了的建筑让人觉得似乎无人照管。于是别人也再不关心。他们放任窗户继续破损。最终自己也参加破坏活动，在外墙上涂鸦，任垃圾堆积。一扇破损的窗户开辟了大厦走向倾颓的道路。

Bjarne也提到完善错误处理代码。往深处说就是在细节上花心思。敷衍了事的错误处理代码只是程序员忽视细节的一种表现。此外还有内存泄漏，还有竞态条件代码。还有前后不一致的命名方式。结果就是凸现出整洁代码对细节的重视。

Bjarne以“整洁的代码只做好一件事”结束论断。毋庸置疑，软件设计的许多原则最终都会归结为这句警语。有那么多人发表过类似的言论。糟糕的代码想做太多事，它意图混乱、目的含混。整洁的代码力求集中。每个函数、每个类和每个模块都全神贯注于一事，完全不受四周细节的干扰和污染。



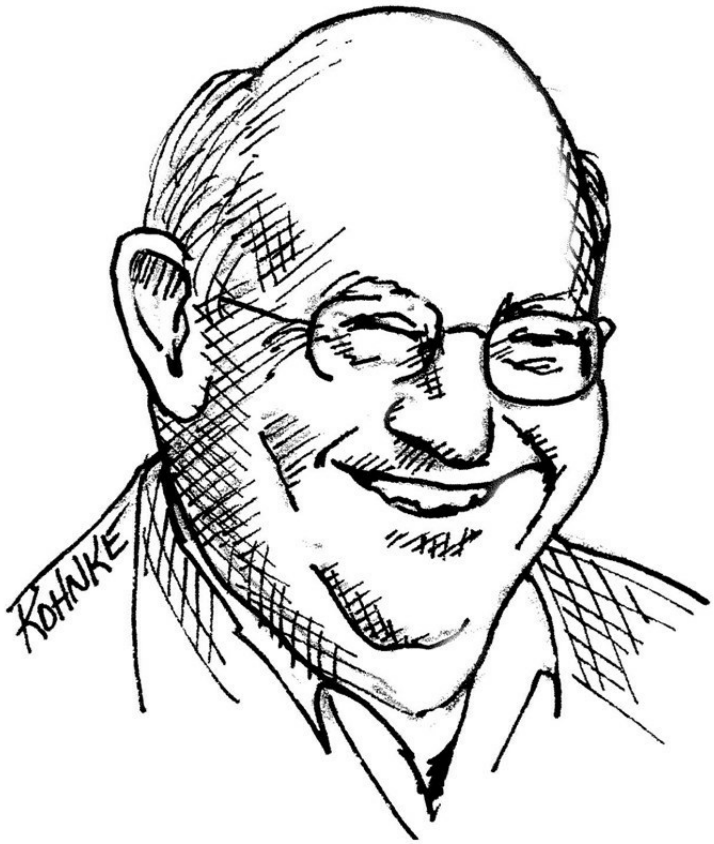
Grady Booch，***Object Oriented Analysis and Design with Applications***（中译版《面向对象分析与设计》）一书作者。

整洁的代码简单直接。整洁的代码如同优美的散文。整洁的代码从不隐藏设计者的意图，充满了干净利落的抽象和直截了当的控制语句。

Grady的观点与Bjarne的观点有类似之处，但他从可读性的角度来定义。我特别喜欢“整洁的代码如同优美的散文”这种看法。想想你读过的某本好书。回忆一下，那些文字是如何在脑中形成影像！就像是看了场电影，对吧？还不止！你还看到那些人物，听到那些声音，体验到那些喜怒哀乐。

阅读整洁的代码和阅读*Lord of the Rings* （中译版《指环王》）自然不同。不过，仍有可类比之处。如同一本好的小说般，整洁的代码应当明确地展现出要解决问题的张力。它应当将这种张力推至高潮，以某种显而易见的方案解决问题和张力，使读者发出“啊哈！本当如此！”的感叹。

窃以为Grady所谓“干净利落的抽象”（crisp abstraction），乃是绝妙的矛盾修辞法。毕竟crisp几乎就是“具体”（concrete）的同义词。我MacBook上的词典这样定义crisp一词：果断决绝，就事论事，没有犹豫或不必要的细节。 尽管有两种不同的定义，该词还是承载了有力的信息。代码应当讲述事实，不引人猜测。它只该包含必需之物。读者应当感受到我们的果断决绝。



“老大**”Dave Thomas**，**OTI**公司创始人，**Eclipse**战略教父。

整洁的代码应可由作者之外的开发者阅读和增补。它应当有单元测试和验收测试。它使用有意义的命名。它只提供一种而非多种做一件事的途径。它只有尽量少的依赖关系，而且要明确地定义和提供清晰、尽量少的API。代码应通过其字面表达含义，因为不同的语言导致并非所有必需信息均可通过代码自身清晰表达。

Dave老大在可读性上和Grady持相同观点，但

有一个重要的不同之处。Dave断言，整洁的代码便于其他人 加以增补。这看似显而易见，但亦不可过分强调。毕竟易读的代码和易修改的代码之间还是有区别的。

Dave将整洁系于测试之上！要在十年之前，这会让人大跌眼镜。但测试驱动开发（Test Driven Development）已在行业中造成了深远影响，成为基础规程之一。Dave说得对。没有测试的代码不干净。不管它有多优雅，不管有多可读、多易理解，微乎测试，其不洁亦可知也。

Dave两次提及“尽量少”。显然，他推崇小块的代码。实际上，从有软件起人们就在反复强调这一点。越小越好。

Dave也提到，代码应在字面上 表达其含义。这一观点源自Knuth的“字面编程”（literate programming）[6] 。结论就是应当用人类可读的方式来写代码。



Michael Feathers，***Working Effectively with Legacy Code***（中译版《修改代码的艺术》）一书作者。

我可以列出我留意到的整洁代码的所有特点，但其中有一条是根本性的。整洁的代码总是看起来像是某位特别在意它的人写的。几乎没有改进的余地。代码作者什么都想到了，如果你企图改进它，总会回到原点，赞叹某人留给你的代码——全心投入的某人留下的代码。

一言以蔽之：在意。这就是本书的题旨所在。或许该加个副标题，如何在意代码 。

Michael一针见血。整洁代码就是作者着力照料的代码。有人曾花时间让它保持简单有序。他们适当地关注到了细节。他们在意过。



Ron Jeffries，***Extreme Programming Installed***（中译版《极限编程实施》）以及***Extreme Programming Adventures in C#***（中译版《**C#**极限编程探险》）作者。

Ron初入行就在战略空军司令部（Strategic Air

Command）编写Fortran程序，此后几乎在每种机器上编写过每种语言的代码。他的言论值得咀嚼。

近年来，我开始研究贝克的简单代码规则，差不多也都琢磨透了。简单代码，依其重要顺序：

- 能通过所有测试；
- 没有重复代码；
- 体现系统中的全部设计理念；
- 包括尽量少的实体，比如类、方法、函数等。

在以上诸项中，我最在意代码重复。如果同一段代码反复出现，就表示某种想法未在代码中得到良好的体现。我尽力去找出到底那是什么，然后再尽力更清晰地表达出来。

在我看来，有意义的命名是体现表达力的一种方式，我往往会修改好几次才会定下名字来。借助Eclipse这样的现代编码工具，重命名代价极低，所以我无所顾忌。然而，表达力还不只体现在命名上。我也会检查对象或方法是否想做的事太多。如果对象功能太多，最好是切分为两个或多个对象。如果方法功能太多，我总是使用抽取手段（Extract Method）重构之，从而得到一个能较为清晰地说明自身功能的方法，以及另外数个说明如何实现这些功能的方法。

消除重复和提高表达力让我在整洁代码方面获益良多，只要铭记这两点，改进脏代码时就会大有不同。不过，我时常关注的另一规则就不太好解释了。

这么多年下来，我发现所有程序都由极为相似的元素构成。例如“在集合中查找某物”。不管是雇员记录数据库还是名-值对哈希表，或者某类条目的数组，我们都会发现自己想要从集合中找到某一特定条目。一旦出现这种情况，我通常会把实现手段封装到更抽象的方法或类中。这样做好处多多。

可以先用某种简单的手段，比如哈希表来实现这一功能，由于对搜索功能的引用指向了我那个小小的抽象，就能随需应变，修改实现手段。这样就既能快速前进，又能为未来的修改预留余地。

另外，该集合抽象常常提醒我留意“真正”在发生的事，避免随意实现集合行为，因为我真正需要的不过是某种简单的查找手段。

减少重复代码，提高表达力，提早构建简单抽象。这就是我写整洁代码的方法。

Ron以寥寥数段文字概括了本书的全部内容。不要重复代码，只做一件事，表达力，小规模抽

象。该有的都有了。



Ward Cunningham，**Wiki**发明者，**eXtreme Programming**（极限编程）的创始人之一，**Smalltalk**语言和面向对象的思想领袖。所有在意代码者的教父。

如果每个例程都让你感到深合己意，那就是整洁代码。如果代码让编程语言看起来像是专为解决那个问题而存在，就可以称之为漂亮的代码。

这种说法很Ward。它教你听了之后就点头，然后继续听下去。如此在理，如此浅显，绝不故作高

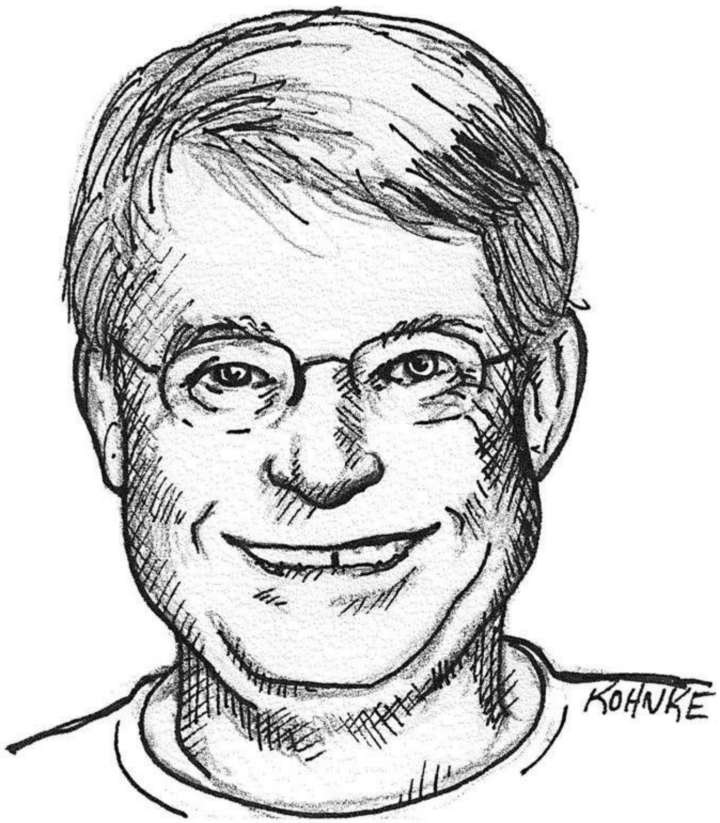
深。你大概以为此言深合己意吧。再走近点看看。

“……深合己意”。你最近一次看到深合己意的模块是什么时候？模块多半都繁复难解吧？难道没有触犯规则吗？你不是也曾挣扎着想抓住些从整个系统中散落而出的线索，编织进你在读的那个模块吗？你最近一次读到某段代码、并且如同对Ward的说法点头一般对这段代码点头，是什么时候的事了？

Ward期望你不会为整洁代码所震惊。你无需花太多力气。那代码就是深合你意。它明确、简单、有力。每个模块都为下一个模块做好准备。每个模块都告诉你下一个模块会是怎样的。整洁的程序好到你根本不会注意到它。设计者把它做得像一切其他设计般简单。

那Ward有关“美”的说法又如何呢？我们都曾面临语言不是为要解决的问题所设计的困境。但Ward的说法又把球踢回我们这边。他说，漂亮的代码让编程语言像是专为解决那个问题而存在！所以，让语言变得简单的责任就在我们 身上了！当心，语言是冥顽不化的！是程序员让语言显得简单。

1.4　思想流派



我（鲍勃大叔）又是怎么想的呢？在我眼中整洁代码是什么样的？本书将以详细到吓死人的程度告诉你，我和我的同道对整洁代码的看法。我们会告诉你关于整洁变量名的想法，关于整洁函数的想法，关于整洁类的想法，如此等等。我们视这些观点为当然，且不为其逆耳而致歉。对我们而言，在职业生涯的这个阶段，这些观点确属当然，也是我们整洁代码派的主旨。

武术家从不认同所谓最好的武术，也不认同所谓绝招。武术大师们常常创建自己的流派，聚徒而授。因此我们才看到格雷西家族在巴西开创并传授的格雷西柔术（Gracie Jiu Jistu），看到奥山龙峰（Okuyama Ryuho）在东京开创并传授的八光流柔术（Hakkoryu Jiu Jistu），看到李小龙（Bruce Lee）在美国开创并传授的截拳道（Jeet Kune Do）。

弟子们沉浸于创始人的授业。他们全心师从某位师傅，排斥其他师傅。弟子有所成就后，可以转投另一位师傅，扩展自己的知识与技能。有些弟子最终百炼成钢，创出新招数，开宗立派。

任何门派都并非绝对正确 。不过，身处某一门派时，我们总以其所传之技为善。归根结底，练习八光流柔术或截拳道，自有其善法，但这并不能否定其他门派所授之法。

可以把本书看作是对象导师 （Object Mentor）[7] 整洁代码派 的说明。里面要传授的就是我们 勤操己艺的方法。如果你遵从这些教诲，你就会如我们一般乐受其益，你将学会如何编写整洁而专业的代码。但无论如何也别错以为我们是“正确的”。其他门派和师傅和我们一样专业。你有必要也向他们学习。

实际上，书中很多建议都存在争议。或许你并不完全同意这些建议。你可能会强烈反对其中一些建议。这样挺好的。我们不能要求做最终权威。另外一方面，书中列出的建议，乃是我们长久苦思、从数十年的从业经验和无数尝试与错误中得来。无论你同意与否，如果你没看到或是不尊敬我们的观点，就真该自己害臊。

1.5　我们是作者

Javadoc中的@author字段告诉我们自己是什么人。我们是作者。作者都有读者。实际上，作者有责任 与读者做良好沟通。下次你写代码的时候，记得自己是作者，要为评判你工作的读者写代码。

你或许会问：代码真正“读”的成分有多少呢？难道力量主要不是用在“写”上吗？

你是否玩过“编辑器回放”？20世纪80、90年代，Emac之类编辑器记录每次击键动作。你可以在一小时工作之后，回放击键过程，就像是看一部高速电影。我这么做过，结果很有趣。

回放过程显示，多数时间都是在滚动屏幕、浏览其他模块！

鲍勃进入模块。

他向下滚动到要修改的函数。

他停下来考虑可以做什么。

哦，他滚动到模块顶端，检查变量初始化。

现在他回到修改处，开始键入。

喔，他删掉了键入的内容。

他重新键入。

他又删除了！

他键入了一半什么东西，又删除掉。

他滚动到调用要修改函数的另一函数，看看是怎么调用的。

他回到修改处，重新键入刚才删掉的代码。

他停下来。

他再一次删掉代码！

他打开另一个窗口，查看别的子类。那是个复载函数吗？

……

你该明白了。读与写花费时间的比例超过10:1。写新代码时，我们一直在读旧代码。

既然比例如此之高，我们就想让读的过程变得轻松，即便那会使得编写过程更难。没可能光写不读，所以使之易读实际也使之易写。

这事概无例外。不读周边代码的话就没法写代码。编写代码的难度，取决于读周边代码的难度。要想干得快，要想早点做完，要想轻松写代码，先让代码易读吧。

1.6 童子军军规

光把代码写好可不够。必须时时保持代码整洁。我们都见过代码随时间流逝而腐坏。我们应当更积极地阻止腐坏的发生。

借用美国童子军一条简单的军规，应用到我们的专业领域：

让营地比你来时更干净。[8]

如果每次签入时，代码都比签出时干净，那么代码就不会腐坏。清理并不一定要花多少功夫，也许只是改好一个变量名，拆分一个有点过长的函数，消除一点点重复代码，清理一个嵌套if语句。

你想要为一个代码随时间流逝而越变越好的项目工作吗？你还能相信有其他更专业的做法吗？难道持续改进不是专业性的内在组成部分吗？

1.7　前传与原则

从许多角度看，本书都是我2002年写那本*Agile Software Development*：*Principles*，*Patterns*，*and Practices*（中译版《敏捷软件开发：原则、模式与实践》，简称PPP）的“前传”。PPP关注面向对象设计的原则，以及专业开发者采用的许多实践方法。假如你没读过PPP，你会发现它像这本书的延续。如果你读过，会发现那本书的主张在代码层面于本书中回响。

在本书中，你会发现对不同设计原则的引用，包括单一权责原则（Single Responsibility Principle，SRP）、开放闭合原则（Open Closed Principle，OCP）和依赖倒置原则（Dependency Inversion Principle，DIP）等。

1.8 小结

艺术书并不保证你读过之后能成为艺术家，只能告诉你其他艺术家用过的工具、技术和思维过程。本书同样也不担保让你成为好程序员。它不担保能给你“代码感”。它所能做的，只是展示好程序员的思维过程，还有他们使用的技巧、技术和工具。

和艺术书一样，本书也充满了细节。代码会很多。你会看到好代码，也会看到糟糕的代码。你会看到糟糕的代码如何转化为好代码。你会看到启发、规条和技巧的列表。你会看到一个又一个例子。但最终结果取决于你自己。

还记得那个关于小提琴家在去表演的路上迷路的老笑话吗？他在街角拦住一位长者，问他怎么才能去卡耐基音乐厅（Carnegie Hall）。长者看了看小提琴家，又看了看他手中的琴，说道：“你还得练，孩子，还得练！”

1.9 文献

[Beck07]： *Implementation Patterns* , Kent Beck, Addison-Wesley, 2007.

[Knuth92]： *Literate Programming* , Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

---

[1] 原注：[Beck07]。

[2] 译注：著名IT讽刺漫画。

[3] 译注：原文为But the fault, dear Dilbert, is not in our stars, but in ourselves.脱胎自莎士比亚戏剧《裘力斯 · 凯撒》第一幕第二场凯些斯的台词The fault, dear Brutus, is not in our stars, but in ourselves, that we are underlings.（若我们受人所制，亲爱的勃鲁托斯，那错也在我们身上，不能怪罪命运。）

[4] 原注：1847年Ignaz Semmelweis（伊纳兹 · 塞麦尔维斯）提出医生应洗手的建议时，遭到了反对，人们认为医生太忙，接诊时无暇洗手。

[5]　原注：http://www.pragmaticprogrammer.com/booksellers/2004-12.html 。

[6]　原注：[Knuth92]。

[7]　译注：本书主要作者Robert C.Martin开办的技术咨询和培训公司。

[8]　原注：摘自Robert Stephenson Smyth Baden-Powell（英国人，童子军创始者）对童子军的遗言："努力，让世界比你来时干净些……"

第2章　有意义的命名

Tim Ottinger



2.1 介绍

软件中随处可见命名。我们给变量、函数、参数、类和封包命名。我们给源代码及源代码所在目录命名。我们给jar文件、war文件和ear文件命名。我们命名、命名，不断命名。既然有这么多命名要做，不妨做好它。下文列出了取个好名字的几条简单规则。

2.2　名副其实

名副其实说起来简单。我们想要强调，这事很严肃 。选个好名字要花时间，但省下来的时间比花掉的多。注意命名，而且一旦发现有更好的名称，就换掉旧的。这么做，读你代码的人（包括你自己）都会更开心。

变量、函数或类的名称应该已经答复了所有的大问题。它该告诉你，它为什么会存在，它做什么事，应该怎么用。如果名称需要注释来补充，那就不算是名副其实。

```
int d;  // 消逝的时间，以日计
```

名称d什么也没说明。它没有引起对时间消逝的感觉，更别说以日计了。我们应该选择指明了计量对象和计量单位的名称：

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

选择体现本意的名称能让人更容易理解和修改代码。下列代码的目的何在？

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
     if (x[0] == 4)
       list1.add(x);
  return list1;
}
```

为什么难以说明上列代码要做什么事？里面并没有复杂的表达式。空格和缩进中规中矩。只用到三个变量和两个常量。甚至没有涉及任何其他类或多态方法，只是（或者看起来是）一个数组的列表而已。

问题不在于代码的简洁度，而是在于代码的模糊度：即上下文在代码中未被明确体现的程度。上列代码要求我们了解类似以下问题的答案：

（1）theList中是什么类型的东西？

（2）theList零下标条目的意义是什么？

（3）值4的意义是什么？

（4）我怎么使用返回的列表？

问题的答案没体现在代码段中，可那就是它们该在 的地方。比方说，我们在开发一种扫雷游戏，我们发现，盘面是名为theList的单元格列表，那就将其名称改为gameBoard。

盘面上每个单元格都用一个简单数组表示。我们还发现，零下标条目是一种状态值，而该种状态值为4表示“已标记”。只要改为有意义的名称，代码就会得到相当程度的改进：

```
public List<int[]> getFlaggedCells()  {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

注意，代码的简洁性并未被触及。运算符和常量的数量全然保持不变，嵌套数量也全然保持不变。但代码变得明确多了。

还可以更进一步，不用int数组表示单元格，而是另写一个类。该类包括一个名副其实的函数（称为isFlagged），从而掩盖住那个魔术数 [1] 。于是得到函数的新版本：

```
public List<Cell> getFlaggedCells()  {
```

```
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

只要简单改一下名称，就能轻易知道发生了什么。这就是选用好名称的力量。

2.3　避免误导

程序员必须避免留下掩藏代码本意的错误线索。应当避免使用与本意相悖的词。例如，hp、aix和sco都不该用做变量名，因为它们都是UNIX平台或类UNIX平台的专有名称。即便你是在编写三角计算程序，hp看起来是个不错的缩写[2]，但那也可能会提供错误信息。

别用accountList来指称一组账号，除非它真的是List类型。List一词对程序员有特殊意义。如果包纳账号的容器并非真是个List，就会引起错误的判断[3]。所以，用accountGroup或bunchOfAccounts，甚至直接用accounts都会好一些。

提防使用不同之处较小的名称。想区分模块中某处的XYZControllerFor EfficientHandlingOfStrings和另一处的XYZControllerForEfficientStorageOfStrings，会花多长时间呢？这两个词外形实在太相似了。

以同样的方式拼写出同样的概念才是信息。拼写前后不一致就是误导 。我们很享受现代Java编程

环境的自动代码完成特性。键入某个名称的前几个字母，按一下某个热键组合（如果有的话），就能得到一列该名称的可能形式。假如相似的名称依字母顺序放在一起，且差异很明显，那就会相当有助益，因为程序员多半会压根不看你的详细注释，甚至不看该类的方法列表就直接看名字挑一个对象。

误导性名称真正可怕的例子，是用小写字母l和大写字母O作为变量名，尤其是在组合使用的时候。当然，问题在于它们看起来完全像是常量“壹”和“零”。

```
int a = l;
if (O == l)
  a = O1;
else
  l = 01;
```

读者可能会认为这纯属虚构，但我们确曾见过充斥这类玩意的代码。有一次，代码作者建议用不同字体写变量名，好显得更清楚些，不过这种方案得要通过口头和书面传递给未来所有的开发者才行。后来，只是做了简单的重命名操作，就解决了问题，而且也没搞出别的事。

2.4 做有意义的区分

如果程序员只是为满足编译器或解释器的需要而写代码，就会制造麻烦。例如，因为同一作用范围内两样不同的东西不能重名，你可能会随手改掉其中一个的名称。有时干脆以错误的拼写充数，结果就是出现在更正拼写错误后导致编译器出错的情况。[4]



光是添加数字系列或是废话远远不够，即便这足以让编译器满意。如果名称必须相异，那其意思也应该不同才对。

以数字系列命名（a1、a2，……aN）是依义命名的对立面。这样的名称纯属误导——完全没有提供正确信息；没有提供导向作者意图的线索。试看：

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

如果参数名改为source和destination，这个函数就会像样许多。

废话是另一种没意义的区分。假设你有一个Product类。如果还有一个ProductInfo或ProductData类，那它们的名称虽然不同，意思却无区别。Info和Data就像a、an和the一样，是意义含混的废话。

注意，只要体现出有意义的区分，使用a和the这样的前缀就没错。例如，你可能把a用在域内变量，而把the用于函数参数[5]。但如果你已经有一个名为zork的变量，又想调用一个名为theZork的变量，麻烦就来了。

废话都是冗余。Variable一词永远不应当出现在变量名中。Table一词永远不应当出现在表名中。

NameString会比Name好吗？难道Name会是一个浮点数不成？如果是这样，就触犯了关于误导的规则。设想有个名为Customer的类，还有一个名为CustomerObject的类。区别何在呢？哪一个是表示客户历史支付情况的最佳途径？

有个应用反映了这种状况。为当事者讳，我们改了一下，不过犯错的代码的确就是这个样子：

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

程序员怎么能知道该调用哪个函数呢？

如果缺少明确约定，变量moneyAmount就与money没区别，customerInfo与customer没区别，accountData与account没区别，theMessage也与message没区别。要区分名称，就要以读者能鉴别不同之处的方式来区分。

2.5 使用读得出来的名称

人类长于记忆和使用单词。大脑的相当一部分就是用来容纳和处理单词的。单词能读得出来。人类进化到大脑中有那么大的一块地方用来处理言语，若不善加利用，实在是种耻辱。

如果名称读不出来，讨论的时候就会像个傻鸟。“哎，这儿，鼻涕阿三喜摁踢（bee cee arr three cee enn tee）[6] 上头，有个皮挨死极翘（pee ess zee kyew）[7] 整数，看见没？”这不是小事，因为编程本就是一种社会活动。

有家公司，程序里面写了个genymdhms（生成日期，年、月、日、时、分、秒），他们一般读作“gen why emm dee aich emm ess”[8]。我有个见字照读的恶习，于是开口就念“gen-yah-mudda-hims”。后来好些设计师和分析师都有样学样，听起来傻乎乎的。我们知道典故，所以会觉得很搞笑。搞笑归搞笑，实际是在强忍糟糕的命名。在给新开发者解释变量的意义时，他们总是读出傻乎乎的自造词，而非恰当的英语词。比较

```
class DtaRcrd102 {
  private Date genymdhms;
```

```
  private Date modymdhms;
  private final String pszqint = "102";
  /*  ...  */
};
```

和

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = "102";
  /*  ...  */
};
```

现在读起来就像人话了：“喂，Mikey，看看这条记录！生成时间戳（generation timestamp）[9] 被设置为明天了！不能这样吧？”

2.6 使用可搜索的名称

单字母名称和数字常量有个问题，就是很难在一大篇文字中找出来。

找MAX_CLASSES_PER_STUDENT很容易，但想找数字7就麻烦了，它可能是某些文件名或其他常量定义的一部分，出现在因不同意图而采用的各种表达式中。如果该常量是个长数字，又被人错改过，就会逃过搜索，从而造成错误。

同样，e也不是个便于搜索的好变量名。它是英文中最常用的字母，在每个程序、每段代码中都有可能出现。由此而见，长名称胜于短名称，搜得到的名称胜于用自造编码代写就的名称。

窃以为单字母名称仅 用于短方法中的本地变量。名称长短应与其作用域大小相对应 [N5]。若变量或常量可能在代码中多处使用，则应赋其以便于搜索的名称。再比较

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

和

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

注意，上面代码中的sum并非特别有用的名称，不过它至少搜得到。采用能表达意图的名称，貌似拉长了函数代码，但要想想看，WORK_DAYS_PER_WEEK要比数字5好找得多，而列表中也只剩下了体现作者意图的名称。

2.7　避免使用编码

编码已经太多，无谓再自找麻烦。把类型或作用域编进名称里面，徒然增加了解码的负担。没理由要求每位新人都在弄清要应付的代码之外（那算是正常的），还要再搞懂另一种编码“语言”。这对于解决问题而言，纯属多余的负担。带编码的名称通常也不便发音，容易打错。

2.7.1　匈牙利语标记法

在往昔名称长短很要命的时代，我们毫无必要地破坏了不编码的规矩，如今后悔不迭。Fortran语言要求首字母体现出类型，导致了编码的产生。BASIC早期版本只允许使用一个字母再加上一位数字。匈牙利语标记法（Hungarian Notation，HN）将这种态势愈演愈烈。

在Windows的C语言API的时代，HN相当重要，那时所有名称要么是个整数句柄，要么是个长指针或者void指针，要不然就是string的几种实现（有不同的用途和属性）之一。那时候编译器并不做类型检查，程序员需要匈牙利语标记法来帮助自己记住类型。

现代编程语言具有更丰富的类型系统，编译器也记得并强制使用类型。而且，人们趋向于使用更小的类、更短的方法，好让每个变量的定义都在视野范围之内。

Java程序员不需要类型编码。对象是强类型的，代码编辑环境已经先进到在编译开始前就侦测到类型错误的程度！所以，如今HN和其他类型编码形式都纯属多余。它们增加了修改变量、函数或类的名称或类型的难度。它们增加了阅读代码的难度。它们制造了让编码系统误导读者的可能性。

```
PhoneNumber  phoneString;
//  类型变化时，名称并不变化！
```

2.7.2　成员前缀

也不必用m_前缀来标明成员变量。应当把类和函数做得足够小，消除对成员前缀的需要。你应当使用某种可以高亮或用颜色标出成员的编辑环境。

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
-----------------------------------------------------
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

此外，人们会很快学会无视前缀（或后缀），只看到名称中有意义的部分。代码读得越多，眼中就越没有前缀。最终，前缀变作了不入法眼的废料，变作了旧代码的标志物。

2.7.3　接口和实现

有时也会出现采用编码的特殊情形。比如，你在做一个创建形状用的抽象工厂（Abstract Factory）。该工厂是个接口，要用具体类来实现。你怎么来命名工厂和具体类呢？IShapeFactory和ShapeFactory吗？我喜欢不加修饰的接口。前导字母I被滥用到了说好听点是干扰，说难听点根本就是废话的程度。我不想让用户知道我给他们的是接口。我就想让他们知道那是个ShapeFactory。如果接口和实现必须选一个来编码的话，我宁肯选择实现。ShapeFactoryImp，甚至是丑陋的CShapeFactory，都比对接口名称编码来得好。

2.8　避免思维映射

不应当让读者在脑中把你的名称翻译为他们熟知的名称。这种问题经常出现在选择是使用问题领域术语还是解决方案领域术语时。

单字母变量名就是个问题。在作用域较小、也没有名称冲突时，循环计数器自然有可能被命名为i或j或k。（但千万别用字母l！）这是因为传统上惯用单字母名称做循环计数器。然而，在多数其他情况下，单字母名称不是个好选择；读者必须在脑中将它映射为真实概念。仅仅是因为有了a和b，就要取名为c，实在并非像样的理由。

程序员通常都是聪明人。聪明人有时会借脑筋急转弯炫耀其聪明。总而言之，假使你记得r代表不包含主机名和图式（scheme）的小写字母版url的话，那你真是太聪明了。

聪明程序员和专业程序员之间的区别在于，专业程序员了解，明确是王道 。专业程序员善用其能，编写其他人能理解的代码。

2.9 类名

类名和对象名应该是名词或名词短语，如Customer、WikiPage、Account和AddressParser。避免使用Manager、Processor、Data或Info这样的类名。类名不应当是动词。

2.10 方法名

方法名应当是动词或动词短语，如postPayment、deletePage或save。属性访问器、修改器和断言应该根据其值命名，并依Javabean标准[10]加上get、set和is前缀。

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

重载构造器时，使用描述了参数的静态工厂方法名。例如，

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

通常好于

```
Complex fulcrumPoint = new  Complex(23.0);
```

可以考虑将相应的构造器设置为private，强制使用这种命名手段。

2.11　别扮可爱

如果名称太耍宝，那就只有同作者一般有幽默感的人才能记得住，而且还是在他们记得那个笑话的时候才行。谁会知道名为HolyHandGrenade [11] 的函数是用来做什么的呢？没错，这名字挺伶俐，不过DeleteItems [12] 或许是更好的名称。宁可明确，毋为好玩。



扮可爱的做法在代码中经常体现为使用俗话或俚语。例如，别用whack( )[13]来表示kill( )。别用eatMyShorts( )[14]这类与文化紧密相关的笑话来表示abort( )。

言到意到。意到言到。

2.12　每个概念对应一个词

给每个抽象概念选一个词，并且一以贯之。例如，使用fetch、retrieve和get来给在多个类中的同种方法命名。你怎么记得住哪个类中是哪个方法呢？很悲哀，你总得记住编写库或类的公司、机构或个人，才能想得起来用的是哪个术语。否则，就得耗费大把时间浏览各个文件头及前面的代码。

Eclipse和IntelliJ之类现代编程环境提供了与环境相关的线索，比如某个对象能调用的方法列表。不过要注意，列表中通常不会给出你为函数名和参数列表编写的注释。如果参数名称来自函数声明，你就太幸运了。函数名称应当独一无二，而且要保持一致，这样你才能不借助多余的浏览就找到正确的方法。

同样，在同一堆代码中有controller，又有manager，还有driver，就会令人困惑。DeviceManager和Protocol-Controller之间有何根本区别？为什么不全用controllers或managers？他们都是Drivers吗？这种名称，让人觉得这两个对象是不同类型的，也分属不同的类。

对于那些会用到你代码的程序员，一以贯之的

命名法简直就是天降福音。

2.13　别用双关语

避免将同一单词用于不同目的。同一术语用于不同概念，基本上就是双关语了。如果遵循“一词一义”规则，可能在好多个类里面都会有add方法。只要这些add方法的参数列表和返回值在语义上等价，就一切顺利。

但是，可能会有人决定为“保持一致”而使用add这个词来命名，即便并非真的想表示这种意思。比如，在多个类中都有add方法，该方法通过增加或连接两个现存值来获得新值。假设要写个新类，该类中有一个方法，把单个参数放到群集（collection）中。该把这个方法叫做add吗？这样做貌似和其他add方法保持了一致，但实际上语义却不同，应该用insert或append之类词来命名才对。把该方法命名为add，就是双关语了。

代码作者应尽力写出易于理解的代码。我们想把代码写得让别人能一目尽览，而不必殚精竭虑地研究。我们想要那种大众化的作者尽责写清楚的平装书模式；我们不想要那种学者挖地三尺才能明白个中意义的学院派模式。

2.14 使用解决方案领域名称

记住，只有程序员才会读你的代码。所以，尽管用那些计算机科学（Computer Science，CS）术语、算法名、模式名、数学术语吧。依据问题所涉领域来命名可不算是聪明的做法，因为不该让协作者老是跑去问客户每个名称的含义，其实他们早该通过另一名称了解这个概念了。

对于熟悉访问者（VISITOR）模式的程序来说，名称AccountVisitor富有意义。哪个程序员会不知道JobQueue的意思呢？程序员要做太多技术性工作。给这些事取个技术性的名称，通常是最靠谱的做法。

2.15 使用源自所涉问题领域的名称

如果不能用程序员熟悉的术语来给手头的工作命名，就采用从所涉问题领域而来的名称吧。至少，负责维护代码的程序员就能去请教领域专家了。

优秀的程序员和设计师，其工作之一就是分离解决方案领域和问题领域的概念。与所涉问题领域更为贴近的代码，应当采用源自问题领域的名称。

2.16　添加有意义的语境

很少有名称是能自我说明的——多数都不能。反之，你需要用有良好命名的类、函数或名称空间来放置名称，给读者提供语境。如果没这么做，给名称添加前缀就是最后一招了。

设想你有名为firstName、lastName、street、houseNumber、city、state和zipcode的变量。当它们搁一块儿的时候，很明确是构成了一个地址。不过，假使只是在某个方法中看见孤零零一个state变量呢？你会理所当然推断那是某个地址的一部分吗？

可以添加前缀addrFirstName、addrLastName、addrState等，以此提供语境。至少，读者会明白这些变量是某个更大结构的一部分。当然，更好的方案是创建名为Address的类。这样，即便是编译器也会知道这些变量隶属某个更大的概念了。

看看代码清单2-1中的方法。以下变量是否需要更有意义的语境呢？函数名仅给出了部分语境；算法提供了剩下的部分。遍览函数后，你会知道number、verb和pluralModifier这三个变量是“测估”信息的一部分。不幸的是这语境得靠读者推断

出来。第一眼看到这个方法时，这些变量的含义完全不清楚。

代码清单2-1　语境不明确的变量

```
private void printGuessStatistics(char candidate, int count) {
  String number;
  String verb;
  String pluralModifier;
  if (count == 0) {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  } else if (count == 1) {
    number = "1";
    verb = "is";
    pluralModifier = "";
  } else {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }
  String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifie
r
  );
  print(guessMessage);
}
```

上列函数有点儿过长，变量的使用贯穿始终。要分解这个函数，需要创建一个名为GuessStatisticsMessage的类，把三个变量做成该类的成员字段。这样它们就在定义上变作了GuessStatisticsMessage的一部分。语境的增强也让算法能够通过分解为更小的函数而变得更为干净利

落。（如代码清单2-2所示。）

代码清单2-2　有语境的变量

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
        verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
```

```
}
```

2.17　不要添加没用的语境

设若有一个名为“加油站豪华版”（Gas Station Deluxe）的应用，在其中给每个类添加GSD前缀就不是什么好点子。说白了，你是在和自己在用的工具过不去。输入G，按下自动完成键，结果会得到系统中全部类的列表，列表恨不得有一英里那么长。这样做聪明吗？为什么要搞得IDE没法帮助你？

再比如，你在GSD应用程序中的记账模块创建了一个表示邮件地址的类，然后给该类命名为GSDAccountAddress。稍后，你的客户联络应用中需要用到邮件地址，你会用GSDAccountAddress吗？这名字听起来没问题吗？在这17个字母里面，有10个字母纯属多余和与当前语境毫无关联。

只要短名称足够清楚，就要比长名称好。别给名称添加不必要的语境。

对于Address类的实体来说，accountAddress和customerAddress都是不错的名称，不过用在类名上就不太好了。Address是个好类名。如果需要与MAC地址、端口地址和Web地址相区别，我会考虑使用PostalAddress、MAC和URI。这样的名称更为

精确，而精确正是命名的要点。

2.18　最后的话

取好名字最难的地方在于需要良好的描述技巧和共有文化背景。与其说这是一种技术、商业或管理问题，还不如说是一种教学问题。其结果是，这个领域内的许多人都没能学会做得很好。

我们有时会怕其他开发者反对重命名。如果讨论一下就知道，如果名称改得更好，那大家真的会感激你。多数时候我们并不记忆类名和方法名。我们使用现代工具对付这些细节，好让自己集中精力于把代码写得就像词句篇章、至少像是表和数据结构（词句并非总是呈现数据的最佳手段）。改名可能会让某人吃惊，就像你做到其他代码改善工作一样。别让这种事阻碍你的前进步伐。

不妨试试上面这些规则，看你的代码可读性是否有所提升。如果你是在维护别人写的代码，使用重构工具来解决问题。效果立竿见影，而且会持续下去。

---

[1]　译注：即表示已标记的4。

[2]　译注：即hypotenuse的缩写。

[3]　原注：如后文提到的，即便容器就是个List，最好也别在名称中写出容器类型名。

[4]　原注：例如，就因为class已有他用，就给一个变量命名为klass，这真是可怕的做法。

[5]　原注：鲍勃大叔惯于在C++中这样做，但后来放弃了，因为现代IDE使这种做法变得没必要了。

[6]　译注：BCR3CNT的读音。

[7]　译注：PSZQ的读音。

[8]　译注：YMDHMS的读音。

[9]　译注：读到generation timestamp时，立刻就能与代码中的generationTimestamp变量对应上。

[10]　原注：http://java.sun.com/products/javabeans/docs/spec.h
。

[11]　译注：意为“圣手手雷”。

[12]　译注：意为“删除条目”。

[13]　美俚，劈砍。

[14] 美俚，去死吧。

第3章　函数



在编程的早年岁月，系统由程序和子程序组成。后来，在Fortran和PL/1的年代，系统由程序、子程序和函数组成。如今，只有函数存活下来。函

数是所有程序中的第一组代码。本章将讨论如何写好函数。

请看代码清单3-1。在FitNesse [1] 中，很难找到长函数，不过我还是搜寻到一个。它不光长，而且代码也很复杂，有大量字符串、怪异而不显见的数据类型和API。花3分钟时间，看能读懂多少？

代码清单3-1　HtmlUtil.java（FitNesse 20070619）

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
               .append(pagePathName)
               .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
```

```
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_TEARDOWN_NAME,
            wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardo
wn);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

搞懂这个函数了吗？大概没有。有太多事发生，有太多不同层级的抽象。奇怪的字符串和函数

调用，混以双重嵌套、用标识来控制的if语句等，不一而足。

不过，只要做几个简单的方法抽离和重命名操作，加上一点点重构，就能在9行代码之内搞掂（如代码清单3-2所示）。用3分钟阅读以下代码，看你能理解吗？

代码清单3-2　HtmlUtil.java（重构之后）

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

除非你正在研究FitNesse，否则就理解不了所有细节。不过，你大概能明白，该函数包含把一些设置和拆解页放入一个测试页面，再渲染为HTML的操作。如果你熟悉JUnit [2]，或许会想到，该函数归属于某个基于Web的测试框架。而且，这当然没错。从代码清单3-2中获得信息很容易，而代码

清单3-1则晦涩难明。

是什么让代码清单3-2易于阅读和理解？怎么才能让函数表达其意图？该给函数赋予哪些属性，好让读者一看就明白函数是属于怎样的程序？

3.1　短小

函数的第一规则是要短小。第二条规则是还要更短小 。我无法证明这个断言。我给不出任何证实了小函数更好的研究结果。我能说的是，近40年来，我写过各种不同大小的函数。我写过令人憎恶的长达3000行的厌物，也写过许多100行到300行的函数，我还写过20行到30行的。经过漫长的试错，经验告诉我，函数就该小。

在20世纪80年代，我们常说函数不该长于一屏。当然，说这话的时候，VT100屏幕只有24行、80列，而编辑器就得先占去4行空间放菜单。如今，用上了精致的字体和宽大的显示器，一屏里面可以显示100行，每行能容纳150个字符。每行都不应该有150个字符那么长。函数也不该有100行那么长，20行封顶最佳。

函数到底该有多长？1991年，我去Kent Beck位于奥勒冈州（Oregon）的家中拜访。我们坐到一起写了些代码。他给我看一个叫做Sparkle（火花闪耀）的有趣的Java/Swing小程序。程序在屏幕上描画电影Cinderella（《灰姑娘》）中仙女用魔棒造出的那种视觉效果。只要移动鼠标，光标所在处就会

爆发出一团令人欣喜的火花，沿着模拟重力场划落到窗口底部。肯特给我看代码的时候，我惊讶于其中那些函数尺寸之小。我看惯了Swing程序中长度数以里计的函数。但这个 程序中每个函数都只有两行、三行或四行长。每个函数都一目了然。每个函数都只说一件事。而且，每个函数都依序把你带到下一个函数。这就是 函数应该达到的短小程度！[3]

函数应该有多短小？通常来说，应该短于代码清单3-2中的函数！代码清单3-2实在应该缩短成代码清单3-3这个样子。

代码清单3-3　HtmlUtil.java（再次重构之后）

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

代码块和缩进

if语句、else语句、while语句等，其中的代码块应该只有一行。该行大抵应该是一个函数调用语句。这样不但能保持函数短小，而且，因为块内调用的函数拥有较具说明性的名称，从而增加了文档

上的价值。

这也意味着函数不应该大到足以容纳嵌套结构。所以，函数的缩进层级不该多于一层或两层。当然，这样的函数易于阅读和理解。

3.2 只做一件事

代码清单3-1显然想做好几件事。它创建缓冲区、获取页面、搜索继承下来的页面、渲染路径、添加神秘的字符串、生成HTML，如此等等。代码清单3-1手忙脚乱。而代码清单3-3则只做一件简单的事。它将设置和拆解包纳到测试页面中。



过去30年以来，以下建议以不同形式一再出现：

函数应该做一件事。做好这件事。只做这一件事。

问题在于很难知道那件该做的事是什么。代码清单3-3只做了一件事，对吧？其实也很容易看作

是三件事：

（1）判断是否为测试页面；

（2）如果是，则容纳进设置和分拆步骤；

（3）渲染成HTML。

那件事是什么？函数是做了一件事呢，还是做了三件事？注意，这三个步骤均在该函数名下的同一抽象层上。可以用简洁的TO [4] 起头段落来描述这个函数：

TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML。

（要*RenderPageWithSetupsAndTeardowns*，检查页面是否为测试页，如果是测试页，就容纳进设置和分拆步骤。无论是否测试页，都渲染成HTML）

如果函数只是做了该函数名下同一抽象层上的步骤，则函数还是只做了一件事。编写函数毕竟是为了把大一些的概念（换言之，函数的名称）拆分为另一抽象层上的一系列步骤。

代码清单3-1明显包括了处于多个不同抽象层级的步骤。显然，它所做的不止一件事。即便是代码清单3-2也有两个抽象层，这已被我们将其缩短的能力所证明。然而，很难再将代码清单3-3做有意义的缩短。可以将if语句拆出来做一个名为includeSetupAndTeardonws IfTestpage的函数，但那只是重新诠释代码，并未改变抽象层级。

所以，要判断函数是否不止做了一件事，还有一个方法，就是看是否能再拆出一个函数，该函数不仅只是单纯地重新诠释其实现[G34]。

函数中的区段

请看代码清单4-7。注意，generatePrimes函数被切分为declarations、initializations和sieve等区段。这就是函数做事太多的明显征兆。只做一件事的函数无法被合理地切分为多个区段。

3.3　每个函数一个抽象层级

要确保函数只做一件事，函数中的语句都要在同一抽象层级上。一眼就能看出，代码清单3-1违反了这条规矩。那里面有getHtml( )等位于较高抽象层的概念，也有String pagePathName = PathParser.render(pagePath)等位于中间抽象层的概念，还有.append("\n")等位于相当低的抽象层的概念。

函数中混杂不同抽象层级，往往让人迷惑。读者可能无法判断某个表达式是基础概念还是细节。更恶劣的是，就像破损的窗户，一旦细节与基础概念混杂，更多的细节就会在函数中纠结起来。

自顶向下读代码：向下规则

我们想要让代码拥有自顶向下的阅读顺序。[5]我们想要让每个函数后面都跟着位于下一抽象层级的函数，这样一来，在查看函数列表时，就能循抽象层级向下阅读了。我把这叫做向下规则 。

换一种说法。我们想要这样读程序：程序就像是一系列TO起头的段落，每一段都描述当前抽象层级，并引用位于下一抽象层级的后续TO起头段

落。

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.（要容纳设置和分拆步骤，就先容纳设置步骤，然后纳入测试页面内容，再纳入分拆步骤。）

To include the setups, we include the suite setup if this is a suite, then we include the regular setup.（要容纳设置步骤，如果是套件，就纳入套件设置步骤，然后再纳入普通设置步骤。）

To include the suite setup, we search the parent hierarchy for the “SuiteSetUp” page and add an include statement with the path of that page.（要容纳套件设置步骤，先搜索“SuiteSetUp”页面的上级继承关系，再添加一个包括该页面路径的语句。）

To search the parent. . .（要搜索……）

程序员往往很难学会遵循这条规则，写出只停留于一个抽象层级上的函数。尽管如此，学习这个技巧还是很重要。这是保持函数短小、确保只做一件事的要诀。让代码读起来像是一系列自顶向下的TO起头段落是保持抽象层级协调一致的有效技

巧。

看看本章末尾的代码清单3-7。它展示了遵循这条原则重构的完整testableHtml函数。留意每个函数是如何引出下一个函数，如何保持在同一抽象层上的。

3.4　switch语句

写出短小的switch语句很难[6]。即便是只有两种条件的switch语句也要比我想要的单个代码块或函数大得多。写出只做一件事的switch语句也很难。Switch天生要做N件事。不幸我们总无法避开switch语句，不过还是能够确保每个switch都埋藏在较低的抽象层级，而且永远不重复。当然，我们利用多态来实现这一点。

请看代码清单3-4。它呈现了可能依赖于雇员类型的仅仅一种操作。

代码清单3-4　Payroll.java

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
  switch (e.type) {
    case COMMISSIONED:
      return calculateCommissionedPay(e);
    case HOURLY:
      return calculateHourlyPay(e);
    case SALARIED:
      return calculateSalariedPay(e);
    default:
      throw new InvalidEmployeeType(e.type);
  }
}
```

该函数有好几个问题。首先，它太长，当出现新的雇员类型时，还会变得更长。其次，它明显做了不止一件事。第三，它违反了单一权责原则（Single Responsibility Principle [7] , SRP），因为有好几个修改它的理由。第四，它违反了开放闭合原则（Open Closed Principle [8] , OCP），因为每当添加新类型时，就必须修改之。不过，该函数最麻烦的可能是到处皆有类似结构的函数。例如，可能会有

```
isPayday(Employee e, Date date),
```

或

```
deliverPay(Employee e, Money pay),
```

如此等等。它们的结构都有同样的问题。

该问题的解决方案（如代码清单3-5所示）是将switch语句埋到抽象工厂 [9] 底下，不让任何人看到。该工厂使用switch语句为Employee的派生物创建适当的实体，而不同的函数，如calculatePay、isPayday和deliverPay等，则藉由Employee接口多态地接受派遣。

对于switch语句，我的规矩是如果只出现一次，用于创建多态对象，而且隐藏在某个继承关系中，在系统其他部分看不到，就还能容忍[G23]。当然也要就事论事，有时我也会部分或全部违反这条规矩。

代码清单3-5　Employee与工厂

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidE
mployeeType;
}
-----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidE
mployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
}
```

3.5 使用描述性的名称

在代码清单3-7中，我把示例函数的名称从testableHtml改为SetupTeardownIncluder.render。这个名称好得多，因为它较好地描述了函数做的事。我也给每个私有方法取个同样具有描述性的名称，如isTestable或includeSetupAndTeardownPages。好名称的价值怎么好评都不为过。记住沃德原则："如果每个例程都让你感到深合己意，那就是整洁代码。"要遵循这一原则，泰半工作都在于为只做一件事的小函数取个好名字。函数越短小、功能越集中，就越便于取个好名字。

别害怕长名称。长而具有描述性的名称，要比短而令人费解的名称好。长而具有描述性的名称，要比描述性的长注释好。使用某种命名约定，让函数名称中的多个单词容易阅读，然后使用这些单词给函数取个能说清其功用的名称。

别害怕花时间取名字。你当尝试不同的名称，实测其阅读效果。在Eclipse或IntelliJ等现代IDE中改名称易如反掌。使用这些IDE测试不同名称，直至找到最具有描述性的那一个为止。

选择描述性的名称能理清你关于模块的设计思

路，并帮你改进之。追索好名称，往往导致对代码的改善重构。

命名方式要保持一致。使用与模块名一脉相承的短语、名词和动词给函数命名。例如，includeSetupAndTeardownPages、includeSetupPages、includeSuiteSetupPage和includeSetupPage等。这些名称使用了类似的措辞，依序讲出一个故事。实际上，假使我只给你看上述函数序列，你就会自问："includeTeardownPages、includeSuiteTeardownPages和includeTeardownPage又会如何？"这就是所谓"深合己意"了。

3.6 函数参数

最理想的参数数量是零（零参数函数），其次是一（单参数函数），再次是二（双参数函数），应尽量避免三（三参数函数）。有足够特殊的理由才能用三个以上参数（多参数函数）——所以无论如何也不要这么做。

参数不易对付。它们带有太多概念性。所以我



参数不易对付。它们带有太多概念性。所以我

在代码范例中几乎不加参数。比如，以StringBuffer为例，我们可能不把它作为实体变量，而是当作参数来传递，那样的话，读者每次看到它都得要翻译一遍。阅读模块所讲述的故事时，includeSetupPage( )要比includeSetupPageInto（newPage-Content）易于理解。参数与函数名处在不同的抽象层级，它要求你了解目前并不特别重要的细节（即那个StringBuffer）。

从测试的角度看，参数甚至更叫人为难。想想看，要编写能确保参数的各种组合运行正常的测试用例，是多么困难的事。如果没有参数，就是小菜一碟。如果只有一个参数，也不太困难。有两个参数，问题就麻烦多了。如果参数多于两个，测试覆盖所有可能值的组合简直让人生畏。

输出参数比输入参数还要难以理解。读函数时，我们惯于认为信息通过参数输入函数，通过返回值从函数中输出 。我们不太期望信息通过参数输出。所以，输出参数往往让人苦思之后才恍然大悟。

相较于没有参数，只有一个输入参数算是第二好的做法。SetupTeardownInclude.render（pageData）也相当易

于理解。很明显，我们将渲染 pageData对象中的数据。

3.6.1　一元函数的普遍形式

向函数传入单个参数有两种极普遍的理由。你也许会问关于那个参数的问题，就像在boolean fileExists("MyFile")中那样。也可能是操作该参数，将其转换为其他什么东西，再输出之。例如，InputStream fileOpen("MyFile")把String类型的文件名转换为InputStream类型的返回值。这就是读者看到函数时所期待的东西。你应当选用较能区别这两种理由的名称，而且总在一致的上下文中使用这两种形式。

还有一种虽不那么普遍但仍极有用的单参数函数形式，那就是事件（event）。在这种形式中，有输入参数而无输出参数。程序将函数看作是一个事件，使用该参数修改系统状态，例如void passwordAttemptFailedNtimes(int attempts)。小心使用这种形式。应该让读者很清楚地了解它是个事件。谨慎地选用名称和上下文语境。

尽量避免编写不遵循这些形式的一元函数，例如，void includeSetupPageInto(StringBuffer pageText)。对于转换，使用输出参数而非返回值令

人迷惑。如果函数要对输入参数进行转换操作，转换结果就该体现为返回值。实际上，StringBuffer transform(StringBuffer in)要比void transform(StringBuffer out)强，即便第一种形式只简单地返回输参数也是这样。至少，它遵循了转换的形式。

3.6.2　标识参数

标识参数丑陋不堪。向函数传入布尔值简直就是骇人听闻的做法。这样做，方法签名立刻变得复杂起来，大声宣布本函数不止做一件事。如果标识为true将会这样做，标识为false则会那样做！

在代码清单3-7中，我们别无选择，因为调用者已经传入了那个标识，而我想把重构范围限制在该函数及该函数以下范围之内。方法调用render(true)对于可怜的读者来说仍然摸不着头脑。卷动屏幕，看到render(Boolean isSuite)，稍许有点帮助，不过仍然不够。应该把该函数一分为二：reanderForSuite( )和renderForSingleTest( )。

3.6.3　二元函数

有两个参数的函数要比一元函数难懂。例如，writeField(name)比writeField(outputStream, name)

[10] 好懂。

尽管两种情况下意义都很清楚，但第一个只要扫一眼就明白，更好地表达了其意义。第二个就得暂停一下才能明白，除非我们学会忽略第一个参数。而且最终那也会导致问题，因为我们根本就不该忽略任何代码。忽略掉的部分就是缺陷藏身之地。

当然，有些时候两个参数正好。例如，Point p = new Point(0,0);就相当合理。笛卡儿点天生拥有两个参数。如果看到new Point(0)，我们会倍感惊讶。然而，本例中的两个参数却只是单个值的有序组成部分！而output-Stream和name则既非自然的组合，也不是自然的排序。

即便是如assertEquals(expected, actual)这样的二元函数也有其问题。你有多少次会搞错actual和expected的位置呢？这两个参数没有自然的顺序。expected在前，actual在后，只是一种需要学习的约定罢了。

二元函数不算恶劣，而且你当然也会编写二元函数。不过，你得小心，使用二元函数要付出代价。你应该尽量利用一些机制将其转换成一元函数。例如，可以把writeField方法写成outputStream

的成员之一，从而能这样用：outputStream.writeField(name)。或者，也可以把outputStream写成当前类的成员变量，从而无需再传递它。还可以分离出类似FieldWriter的新类，在其构造器中采用outputStream，并且包含一个write方法。

3.6.4 三元函数

有三个参数的函数要比二元函数难懂得多。排序、琢磨、忽略的问题都会加倍体现。建议你在写三元函数前一定要想清楚。

例如，设想assertEquals有三个参数：assertEquals(message, expected, actual)。有多少次，你读到message，错以为它是expected呢？我就常栽在这个三元函数上。实际上，每次我看到这里，总会绕半天圈子，最后学会了忽略message参数。

另一方面，这里有个并不那么险恶的三元函数：assertEquals(1.0, amount, .001)。虽然也要费点神，还是值得的。得到“浮点值的等值是相对而言”的提示总是好的。

3.6.5 参数对象

如果函数看来需要两个、三个或三个以上参数，就说明其中一些参数应该封装为类了。例如，下面两个声明的差别：

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

从参数创建对象，从而减少参数数量，看起来像是在作弊，但实则并非如此。当一组参数被共同传递，就像上例中的x和y那样，往往就是该有自己名称的某个概念的一部分。

3.6.6　参数列表

有时，我们想要向函数传入数量可变的参数。例如，String.format方法：

```
String.format("%s worked %.2f hours.", name, hours);
```

如果可变参数像上例中那样被同等对待，就和类型为List的单个参数没什么两样。这样一来，String.formate实则是二元函数。下列String.format的声明也很明显是二元的：

```
public String format(String format, Object... args)
```

同理，有可变参数的函数可能是一元、二元甚至三元。超过这个数量就可能要犯错了。

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

3.6.7 动词与关键字

给函数取个好名字，能较好地解释函数的意图，以及参数的顺序和意图。对于一元函数，函数和参数应当形成一种非常良好的动词/名词对形式。例如，write(name)就相当令人认同。不管这个“name”是什么，都要被“write”。更好的名称大概是writeField(name)，它告诉我们，“name”是一个“field”。

最后那个例子展示了函数名称的关键字（keyword）形式。使用这种形式，我们把参数的名称编码成了函数名。例如，assertEqual改成assertExpectedEqualsActual(expected, actual)可能会好些。这大大减轻了记忆参数顺序的负担。

3.7　无副作用

副作用是一种谎言。函数承诺只做一件事，但还是会做其他被藏起来的事。有时，它会对自己类中的变量做出未能预期的改动。有时，它会把变量搞成向函数传递的参数或是系统全局变量。无论哪种情况，都是具有破坏性的，会导致古怪的时序性耦合及顺序依赖。

以代码清单3-6中看似无伤大雅的函数为例。该函数使用标准算法来匹配userName和password。如果匹配成功，返回true，如果失败则返回false。但它会有副作用。你知道问题所在吗？

代码清单3-6　UserValidator.java

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password)
 {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, passwor
d);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
```

```
  }
}
```

当然了，副作用就在于对Session.initialize( )的调用。checkPassword函数，顾名思义，就是用来检查密码的。该名称并未暗示它会初始化该次会话。所以，当某个误信了函数名的调用者想要检查用户有效性时，就得冒抹除现有会话数据的风险。

这一副作用造出了一次时序性耦合。也就是说，checkPassword只能在特定时刻调用（换言之，在初始化会话是安全的时候调用）。如果在不合适的时候调用，会话数据就有可能沉默地丢失。时序性耦合令人迷惑，特别是当它躲在副作用后面时。如果一定要时序性耦合，就应该在函数名称中说明。在本例中，可以重命名函数为checkPasswordAndInitializeSession，虽然那还是违反了“只做一件事”的规则。

输出参数

参数多数会被自然而然地看作是函数的输入。如果你编过好些年程序，我担保你一定被用作输出而非输入的参数迷惑过。例如：

```
appendFooter(s);
```

这个函数是把s添加到什么东西后面吗？或者它把什么东西添加到了s后面？s是输入参数还是输出参数？稍许花点时间看看函数签名：

```
public void appendFooter(StringBuffer report)
```

事情清楚了，但付出了检查函数声明的代价。你被迫检查函数签名，就得花上一点时间。应该避免这种中断思路的事。

在面向对象编程之前的岁月里，有时的确需要输出参数。然而，面向对象语言中对输出参数的大部分需求已经消失了，因为this也有输出函数的意味在内。换言之，最好是这样调用appendFooter：

```
report.appendFooter();
```

普遍而言，应避免使用输出参数。如果函数必须要修改某种状态，就修改所属对象的状态吧。

3.8　分隔指令与询问

函数要么做什么事，要么回答什么事，但二者不可得兼。函数应该修改某对象的状态，或是返回该对象的有关信息。两样都干常会导致混乱。看看下面的例子：

```
public boolean set(String attribute, String value);
```

该函数设置某个指定属性，如果成功就返回true，如果不存在那个属性则返回false。这样就导致了以下语句：

```
if (set("username", "unclebob"))...
```

从读者的角度考虑一下吧。这是什么意思呢？它是在问username属性值是否之前已设置为unclebob吗？或者它是在问username属性值是否成功设置为unclebob呢？从这行调用很难判断其含义，因为set是动词还是形容词并不清楚。

作者本意，set是个动词，但在if语句的上下文中，感觉 它像是个形容词。该语句读起来像是

说“如果username属性值之前已被设置为uncleob”，而不是“设置username属性值为unclebob，看看是否可行，然后……”。要解决这个问题，可以将set函数重命名为setAndCheckIfExists，但这对提高if语句的可读性帮助不大。真正的解决方案是把指令与询问分隔开来，防止混淆的发生：

```
if (attributeExists("username")) {
   setAttribute("username", "unclebob");
   ...
}
```

3.9 使用异常替代返回错误码

从指令式函数返回错误码轻微违反了指令与询问分隔的规则。它鼓励了在if语句判断中把指令当作表达式使用。

```
if (deletePage(page) == E_OK)
```

这不会引起动词/形容词混淆，但却导致更深层次的嵌套结构。当返回错误码时，就是在要求调用者立刻处理错误。

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

另一方面，如果使用异常替代返回错误码，错

误处理代码就能从主路径代码中分离出来，得到简化：

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

3.9.1　抽离Try/Catch代码块

Try/catch代码块丑陋不堪。它们搞乱了代码结构，把错误处理与正常流程混为一谈。最好把try和catch代码块的主体部分抽离出来，另外形成函数。

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Excep
tion {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
```

```
  logger.log(e.getMessage());
}
```

在上例中，delete函数只与错误处理有关。很容易理解然后就忽略掉。deletePageAndAllReference函数只与完全删除一个page有关。错误处理可以忽略掉。有了这样美妙的区隔，代码就更易于理解和修改了。

3.9.2 错误处理就是一件事

函数应该只做一件事。错误处理就是一件事。因此，处理错误的函数不该做其他事。这意味着（如上例所示）如果关键字try在某个函数中存在，它就该是这个函数的第一个单词，而且在catch/finally代码块后面也不该有其他内容。

3.9.3 Error.java依赖磁铁

返回错误码通常暗示某处有个类或是枚举，定义了所有错误码。

```
public enum Error {
   OK,
   INVALID,
   NO_SUCH,
   LOCKED,
   OUT_OF_RESOURCES,
```

```
   WAITING_FOR_EVENT;
}
```

这样的类就是一块依赖磁铁（dependency magnet）；其他许多类都得导入和使用它。当Error枚举修改时，所有这些其他的类都需要重新编译和部署。[11] 这对Error类造成了负面压力。程序员不愿增加新的错误代码，因为这样他们就得重新构建和部署所有东西。于是他们就复用旧的错误码，而不添加新的。

使用异常替代错误码，新异常就可以从异常类派生 出来，无需重新编译或重新部署[12] 。

3.10 别重复自己[13]

回头仔细看看代码清单3-1，你会注意到，有个算法在SetUp、SuiteSetUp、TearDown和SuiteTearDown中总共被重复了4次。识别重复不太容易，因为这4次重复与其他代码混在一起，而且也不完全一样。这样的重复还是会导致问题，因为代码因此而臃肿，且当算法改变时需要修改4处地方。而且也会增加4次放过错误的可能性。



使用代码清单3-7中的include方法修正了这些重复。再读一遍那段代码，你会注意到，整个模块的可读性因为重复的消除而得到了提升。

重复可能是软件中一切邪恶的根源。许多原则与实践规则都是为控制与消除重复而创建。例如，全部考德（Codd）[14]数据库范式都是为消灭数据重复而服务。再想想看，面向对象编程是如何将代码集中到基类，从而避免了冗余。面向方面编程（Aspect Oriented Programming）、面向组件编程（Component Oriented Programming）多少也都是消除重复的一种策略。看来，自子程序发明以来，软

件开发领域的所有创新都是在不断尝试从源代码中消灭重复。

3.11　结构化编程

有些程序员遵循Edsger Dijkstra的结构化编程规则[15]。Dijkstra认为，每个函数、函数中的每个代码块都应该有一个入口、一个出口。遵循这些规则，意味着在每个函数中只该有一个return语句，循环中不能有break或continue语句，而且永永远远不能有任何goto语句。

我们赞成结构化编程的目标和规范，但对于小函数，这些规则助益不大。只有在大函数中，这些规则才会有明显的好处。

所以，只要函数保持短小，偶尔出现的return、break或continue语句没有坏处，甚至还比单入单出原则更具有表达力。另外一方面，goto只在大函数中才有道理，所以应该尽量避免使用。

3.12　如何写出这样的函数

写代码和写别的东西很像。在写论文或文章时，你先想什么就写什么，然后再打磨它。初稿也许粗陋无序，你就斟酌推敲，直至达到你心目中的样子。

我写函数时，一开始都冗长而复杂。有太多缩进和嵌套循环。有过长的参数列表。名称是随意取的，也会有重复的代码。不过我会配上一套单元测试，覆盖每行丑陋的代码。

然后我打磨这些代码，分解函数、修改名称、消除重复。我缩短和重新安置方法。有时我还拆散类。同时保持测试通过。

最后，遵循本章列出的规则，我组装好这些函数。

我并不从一开始就按照规则写函数。我想没人做得到。

3.13 小结

每个系统都是使用某种领域特定语言搭建，而这种语言是程序员设计来描述那个系统的。函数是语言的动词，类是名词。这并非是退回到那种认为需求文档中的名词和动词就是系统中类和函数的最初设想的可怕的旧观念。其实这是个历史更久的真理。编程艺术是且一直就是语言设计的艺术。

大师级程序员把系统当作故事来讲，而不是当作程序来写。他们使用选定编程语言提供的工具构建一种更为丰富且更具表达力的语言，用来讲那个故事。那种领域特定语言的一个部分，就是描述在系统中发生的各种行为的函数层级。在一种狡猾的递归操作中，这些行为使用它们定义的与领域紧密相关的语言讲述自己那个小故事。

本章所讲述的是有关编写良好函数的机制。如果你遵循这些规则，函数就会短小，有个好名字，而且被很好地归置。不过永远别忘记，真正的目标在于讲述系统的故事，而你编写的函数必须干净利落地拼装到一起，形成一种精确而清晰的语言，帮助你讲故事。

3.14　SetupTeardownIncluder程序

代码清单3-7　SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;

  public static String render(PageData pageData) throws Exceptio
n {
    return render(pageData, false);
}

  public static String render(PageData pageData, boolean isSuite
)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }
  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }
```

```
private boolean isTestPage() throws Exception {
  return pageData.hasAttribute("Test");
}

private void includeSetupAndTeardownPages() throws Exception {
  includeSetupPages();
  includePageContent();
  includeTeardownPages();
  updatePageContent();
}

private void includeSetupPages() throws Exception {
  if (isSuite)
    includeSuiteSetupPage();
  includeSetupPage();
}

private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
```

```
  }

  private void include(String pageName, String arg) throws Excep
tion {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
      String pagePathName = getPathNameForPage(inheritedPage);
      buildIncludeDirective(pagePathName, arg);
    }
  }

  private WikiPage findInheritedPage(String pageName) throws Exc
eption {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
  }

  private String getPathNameForPage(WikiPage page) throws Except
ion {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String
 arg) {
    newPageContent
    .append("\n!include ")
    .append(arg)
    .append(" .")
    .append(pagePathName)
    .append("\n");
  }
}
```

3.15 文献

[KP78]：Kernighan and Plaugher, *The Elements of Programming Style*, 2d. ed., McGraw- Hill, 1978.

[PPP02]：Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[PRAG]：*The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[SP72]：*Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.

---

[1] 原注：一种开源测试工具。见http://www.fitnese.org 。

[2] 原注：一种开源Java单元测试工具。见

http://www.junit.org 。

[3]　原注：我问肯特是否还保留这段程序，他说找不到了。我搜遍自己的电脑也没找到。现在只有在记忆中有这段程序了。

[4]　原注：LOGO语言中的TO关键字，与Ruby和Python中def关键字的用法一致。所以，每个函数都以TO起头。这对函数的设计产生了有趣的影响。

[5]　原注：[KP78]。

[6]　原注：当然，这也包括if/else语句在内。

[7]　原注：a. http://en.wikipedia.org/wiki/Single_responsibility_princ ；
b. http://www.objectmentor.com/resources/articles/srp.pdf 。

[8]　原注：a. http://en.wikipedia.org/wiki/Open/closed_principle ；
b. http://www.objectmentor.com/resources/articles/ocp.pd 。

[9]　原注：[GOF]。

[10]　原注：我刚重构了一个使用了二元形式的模块。现在就能把outputStream做成该类的一个字段，并把所有对writeField的调用都变作一元形式。结果就干净多了。

[11]　原注：那些以为可以不重新编译和部署就扬长而去的家伙最终都自尝恶果。

[12]　原注：这也是开放闭合原则（OCP）的一个范例 [PPP02]。

[13]　原注：DRY原则。[PRAG]。

[14]　译注：艾德加 · F · 考德（Edgar F. Codd），关系数据库之父。

[15]　原注：[SP72]。

第4章　注释



“别给糟糕的代码加注释——重新写吧。”

—Brian W. Kernighan与P. J. Plaugher [1]

什么也比不上放置良好的注释来得有用。什么也不会比乱七八糟的注释更有本事搞乱一个模块。什么也不会比陈旧、提供错误信息的注释更有破坏性。

注释并不像辛德勒的名单。它们并不“纯然地好”。实际上，注释最多也就是一种必须的恶。若编程语言足够有表达力，或者我们长于用这些语言来表达意图，就不那么需要注释——也许根本不需要。

注释的恰当用法是弥补我们在用代码表达意图时遭遇的失败。注意，我用了“失败 ”一词。我是说真的。注释总是一种失败。我们总无法找到不用注释就能表达自我的方法，所以总要有注释，这并不值得庆贺。

如果你发现自己需要写注释，再想想看是否有办法翻盘，用代码来表达。每次用代码表达，你都该夸奖一下自己。每次写注释，你都该做个鬼脸，感受自己在表达能力上的失败。

我为什么要极力贬低注释？因为注释会撒谎。也不是说总是如此或有意如此，但出现得实在太频繁。注释存在的时间越久，就离其所描述的代码越远，越来越变得全然错误。原因很简单。程序员不能坚持维护注释。

代码在变动，在演化。从这里移到那里。彼此分离、重造又合到一处。很不幸，注释并不总是随之变动——不能 总是跟着走。注释常常会与其所描述的代码分隔开来，孑然飘零，越来越不准确。例如，看看以下注释以及它本来要描述的代码行变成了什么样子：

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
```

```
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

在HTTP_DATE_REGEXP常量及其注释之间，有可能插入其他实体变量。

程序员应当负责将注释保持在可维护、有关联、精确的高度。我同意这种说法。但我更主张把力气用在写清楚代码上，直接保证无须编写注释。

不准确的注释要比没注释坏得多。它们满口胡言。它们预期的东西永不能实现。它们设定了无需也不应再遵循的旧规则。

真实只在一处地方有：代码。只有代码能忠实地告诉你它做的事。那是唯一真正准确的信息来源。所以，尽管有时也需要注释，我们也该多花心思尽量减少注释量。

4.1　注释不能美化糟糕的代码

写注释的常见动机之一是糟糕的代码的存在。我们编写一个模块，发现它令人困扰、乱七八糟。我们知道，它烂透了。我们告诉自己：“喔，最好写点注释！”不！最好是把代码弄干净！

带有少量注释的整洁而有表达力的代码，要比带有大量注释的零碎而复杂的代码像样得多。与其花时间编写解释你搞出的糟糕的代码的注释，不如花时间清洁那堆糟糕的代码。

4.2 用代码来阐述

有时，代码本身不足以解释其行为。不幸的是，许多程序员据此以为代码很少——如果有的话——能做好解释工作。这种观点纯属错误。你愿意看到这个：

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

还是这个？

```
if (employee.isEligibleForFullBenefits())
```

只要想上那么几秒钟，就能用代码解释你大部分的意图。很多时候，简单到只需要创建一个描述与注释所言同一事物的函数即可。

4.3 好注释

有些注释是必须的，也是有利的。来看看一些我认为值得写的注释。不过要记住，唯一真正好的注释是你想办法不去写的注释。

4.3.1 法律信息

有时，公司代码规范要求编写与法律有关的注释。例如，版权及著作权声明就是必须和有理由在每个源文件开头注释处放置的内容。

下例是我们在FitNesse项目每个源文件开头放置的标准注释。我可以很开心地说，IDE自动卷起这些注释，这样就不会显得凌乱了。

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All righ
ts reserved.
// Released under the terms of the GNU General Public License v
ersion 2 or later.
```

这类注释不应是合同或法典。只要有可能，就指向一份标准许可或其他外部文档，而不要把所有条款放到注释中。

4.3.2　提供信息的注释

有时，用注释来提供基本信息也有其用处。例如，以下注释解释了某个抽象方法的返回值：

```
// Returns an instance of the Responder being tested.

protected abstract Responder responderInstance();
```

这类注释有时管用，但更好的方式是尽量利用函数名称传达信息。比如，在本例中，只要把函数重新命名为responderBeingTested，注释就是多余的了。

下例稍好一些：

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

在本例中，注释说明，该正则表达式意在匹配一个经由SimpleDateFormat.format函数利用特定格式字符串格式化的时间和日期。同样，如果把这段代码移到某个转换日期和时间格式的类中，就会更

好、更清晰，而注释也就变得多此一举了。

4.3.3　对意图的解释

　　有时，注释不仅提供了有关实现的有用信息，而且还提供了某个决定后面的意图。在下例中，我们看到注释反映出来的一个有趣决定。在对比两个对象时，作者决定将他的类放置在比其他东西更高的位置。

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "")
;
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // we are greater because we are the right type.



}
```

　　下面的例子甚至更好。你也许不同意程序员给这个问题提供的解决方案，但至少你知道他想干什么。

```
public void testConcurrentAddWidgets() throws Exception {
```

```
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
  String text = "'''bold text'''";
  ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
  AtomicBoolean failFlag = new AtomicBoolean();
  failFlag.set(false);

 //This is our best attempt to get a race condition



  //by creating large number of threads.



  for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
      new WidgetBuilderThread(widgetBuilder, text, parent, failF
lag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
  }
  assertEquals(false, failFlag.get());
}
```

4.3.4　阐释

有时，注释把某些晦涩难明的参数或返回值的意义翻译为某种可读形式，也会是有用的。通常，更好的方法是尽量让参数或返回值自身就足够清楚；但如果参数或返回值是某个标准库的一部分，或是你不能修改的代码，帮助阐释其含义的代码就会有用。

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

当然，这也会冒阐释性注释本身就不正确的风险。回头看看上例，你会发现想要确认注释的正确性有多难。这一方面说明了阐释有多必要，另外也说明了它有风险。所以，在写这类注释之前，考虑一下是否还有更好的办法，然后再加倍小心地确认注释正确性。

4.3.5 警示

有时，用于警告其他程序员会出现某种后果的注释也是有用的。例如，下面的注释解释了为什么要关闭某个特定的测试用例：

```
//  Don't  run  unless  you

//  have  some  time  to  kill.

public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);

  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```



当然，如今我们多数会利用附上恰当解释性字符串的@Ignore属性来关闭测试用例。比如@Ignore("Takes too long to run [2] ")。但在JUnit4之前的日子里，惯常的做法是在方法名前面加上下划线。如果注释足够有说服力，就会很有用了。

这里有个更麻烦的例子：

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
```

```
  //SimpleDateFormat is not thread safe,

  //so we need to create each instance independently.

  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy
HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

你也许会抱怨说，还会有更好的解决方法。我大概会同意。不过上面的注释绝对有道理存在，它能阻止某位急切的程序员以效率之名使用静态初始器。

4.3.6　TODO注释

有时，有理由用//TODO形式在源代码中放置要做的工作列表。在下例中，TODO注释解释了为什么该函数的实现部分无所作为，将来应该是怎样。

```
//TODO-MdM these are not needed

//  We expect this to go away when we do the checkout model
```

```
protected VersionInfo makeVersion() throws Exception
{
return null;
}
```

TODO是一种程序员认为应该做，但由于某些原因目前还没做的工作。它可能是要提醒删除某个不必要的特性，或者要求他人注意某个问题。它可能是恳请别人取个好名字，或者提示对依赖于某个计划事件的修改。无论TODO的目的如何，它都不是在系统中留下糟糕的代码的借口。

如今，大多数好IDE都提供了特别的手段来定位所有TODO注释，这些注释看来丢不了。你不会愿意代码因为TODO的存在而变成一堆垃圾，所以要定期查看，删除不再需要的。

4.3.7　放大

注释可以用来放大某种看来不合理之物的重要性。

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
```

```
// spaces that could cause the item to be recognized

// as another list.

new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

4.3.8　公共API中的Javadoc

没有什么比被良好描述的公共API更有用和令人满意的了。标准Java库中的Javadoc就是一例。没有它们，写Java程序就会变得很难。

如果你在编写公共API，就该为它编写良好的Javadoc。不过要记住本章中的其他建议。就像其他注释一样，Javadoc也可能误导、不适用或者提供错误信息。

4.4 坏注释

大多数注释都属此类。通常，坏注释都是糟糕的代码的支撑或借口，或者对错误决策的修正，基本上等于程序员自说自话。

4.4.1 喃喃自语

如果只是因为你觉得应该或者因为过程需要就添加注释，那就是无谓之举。如果你决定写注释，就要花必要的时间确保写出最好的注释。

例如，我在FitNesse中找到的这个例子，例中的注释大概确实有用。不过，作者太着急，或者没太花心思。他的喃喃自语变成了一个谜团。

```
public void loadProperties()
{
  try
  {
    String propertiesPath = propertiesLocation + "/" + PROPERTIE
S_FILE;
    FileInputStream propertiesStream = new FileInputStream(prope
rtiesPath);
    loadedProperties.load(propertiesStream);
  }
  catch(IOException e)
  {
    // No properties files means all defaults are loaded
```

```
  }
}
```

catch代码块中的注释是什么意思呢？显然对于作者有其意义，不过并没有好到足够的程度。很明显，如果出现IOException，就表示没有属性文件；在那种情况下，载入默认设置。但谁来装载默认设置呢？会在对loadProperties.load之前装载吗？抑或loadProperties.load捕获异常、装载默认设置、再向上传递异常？再或loadProperties.load在尝试载入文件前就装载所有默认设置？要么作者只是在安慰自己别在意catch代码块的留空？或者——这种可能最可怕——作者是想告诉自己，将来再回头写装载默认设置的代码？

我们唯有检视系统其他部分的代码，弄清事情原委。任何迫使读者查看其他模块的注释，都没能与读者沟通好，不值所费。

4.4.2　多余的注释

代码清单4-1展示的简单函数，其头部位置的注释全属多余。读这段注释花的时间没准比读代码花的时间还要长。

代码清单4-1　waitForClose

```
// Utility method that returns when this.closed is true. Throws
 an exception



// if the timeout is reached.



public synchronized void waitForClose(final long timeoutMillis)

throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be close
d");
  }
}
```

这段注释起了什么作用？它并不能比代码本身提供更多的信息。它没有证明代码的意义，也没有给出代码的意图或逻辑。读它并不比读代码更容易。事实上，它不如代码精确，误导读者接受不精确的信息，而不是正确地理解代码。它就像个自来熟的二手车贩子，满口保证你不用打开发动机盖查验。

来看看代码清单4-2中摘自Tomcat项目的无用

而多余的Javadoc吧。这些注释只是一味将代码搞得含糊不明。完全没有文档上的价值。下面只列出了靠前面的一些代码，后续模块中还有许多类似情况。

代码清单4-2　ContainerBase.java　（Tomcat）

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * The processor delay for this component.
   */
  protected int backgroundProcessorDelay = -1;

  /**
   * The lifecycle event support for this component.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);

  /**
   * The container event listeners for this Container.
   */
  protected ArrayList listeners = new ArrayList();

  /**
   * The Loader implementation with which this Container is
   * associated.
   */
  protected Loader loader = null;

  /**
   * The Logger implementation with which this Container is
   * associated.
   */
  protected Log logger = null;

  /**
   * Associated logger name.
```

```
 */
protected String logName = null;

/**
 * The Manager implementation with which this Container is
 * associated.
 */
protected Manager manager = null;

/**
 * The cluster with which this Container is associated.
 */
protected Cluster cluster = null;

/**
 * The human-readable name of this Container.
 */
protected String name = null;

/**
 * The parent Container to which this Container is a child.
 */
protected Container parent = null;

/**
 * The parent class loader to be configured when we install a
 * Loader.
 */
protected ClassLoader parentClassLoader = null;

/**
 * The Pipeline object with which this Container is
 * associated.
 */
protected Pipeline pipeline = new StandardPipeline(this);

/**
 * The Realm with which this Container is associated.
 */
protected Realm realm = null;

/**
 * The resources DirContext object with which this Container
 * is associated.
 */
```

```
protected DirContext resources = null;
```

4.4.3　误导性注释

有时，尽管初衷可嘉，程序员还是会写出不够精确的注释。想想看代码清单4-1中那多余而又有误导嫌疑的注释吧。

你有没有发现那样的注释是如何误导读者的？在this.closed变为true的时候，方法并没有返回。方法只在判断到this.closed为true的时候返回，否则，就只是等待遥遥无期的超时，然后如果判断this.closed还是非true，就抛出一个异常。

这一细微的误导信息，放在比代码本身更难阅读的注释里面，有可能导致其他程序员快活地调用这个函数，并期望在this.closed变为true时立即返回。那位可怜的程序员将会发现自己陷于调试困境之中，拼命想找出代码执行得如此之慢的原因。

4.4.4　循规式注释

所谓每个函数都要有Javadoc或每个变量都要有注释的规矩全然是愚蠢可笑的。这类注释徒然让代码变得散乱，满口胡言，令人迷惑不解。

例如，要求每个函数都要有Javadoc，就会得到类似代码清单4-3那样面目可憎的代码。这类废话只会搞乱代码，有可能误导读者。

代码清单4-3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

4.4.5 日志式注释

有人会在每次编辑代码时，在模块开始处添加一条注释。这类注释就像是一种记录每次修改的日志。我见过满篇尽是这类日志的代码模块。

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new pack
age
*               com.jrefinery.date (DG);
```

```
* 05-Nov-2001 : Added a getDescription() method, and eliminated
 NotableDate
*               class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that
NotableDate
*               class is gone (DG);  Changed getPreviousDayOfWee
k(),
*               getFollowingDayOfWeek() and getNearestDayOfWeek
() to correct
*               bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate inter
face
*              (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???
levka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 04-Sep-2003 : Implemented Comparable.  Updated the isInRange j
avadocs (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

很久以前，在模块开始处创建并维护这些记录还算有道理。那时，我们还没有源代码控制系统可用。如今，这种冗长的记录只会让模块变得凌乱不堪，应当全部删除。

4.4.6 废话注释

有时，你会看到纯然是废话的注释。它们对于显然之事喋喋不休，毫无新意。

```
/**
*  Default constructor.
```

```
*/
protected AnnualDateRule() {
}
```

对吧？再看看这个：

```
/** The day of the month. */
    private int dayOfMonth;
```

还有这样的废话模范：

```
/**
* Returns the day of the month.
*
* @return the day of the month.
*/
public int getDayOfMonth() {
  return dayOfMonth;
}
```

这类注释废话连篇，我们都学会了视而不见。读代码时，眼光不会停留在它们上面。最终，当代码修改之后，这类注释就变作了谎言一堆。

代码清单4-4中的第一条注释貌似还行 [3] 。它解释了catch代码块为何被忽略。不过第二条注释就纯是废话了。显然，该程序员沮丧于编写函数中那些try/catch代码块。

代码清单4-4　startSending

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // normal. someone stopped the request.



  }
  catch(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
      // Give me a break!



    }
  }
}
```

与其纠缠于毫无价值的废话注释，程序员应该意识到，他的挫败感可以由改进代码结构而消除。他应该把力气花在将最末一个try/catch代码块拆解到单独的函数中，如代码清单4-5所示。

代码清单4-5　startSending（重构之后）

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    //normal. someone stopped the request.
  }
  catch(Exception e)
  {
    addExceptionAndCloseResponse(e);
  }
}

private void addExceptionAndCloseResponse(Exception e)
{
  try
  {
    response.add(ErrorResponder.makeExceptionString(e));
    response.closeAll();
  }
  catch(Exception e1)
  {
  }
}
```

用整理代码的决心替代创造废话的冲动吧。你会发现自己成为更优秀、更快乐的程序员。

4.4.7　可怕的废话

Javadoc也可能是废话。下列Javadoc（来自某

知名开源库）的目的是什么？答案：无。它们只是源自某种提供文档的不当愿望的废话注释。

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The version. */
private String info;
```

再仔细读读这些注释。你是否发现了剪切-粘贴错误？如果作者在写（或粘贴）注释时都没花心思，怎么能指望读者从中获益呢？

4.4.8 能用函数或变量时就别用注释

看看以下代码概要：

```
// does the module from the global list <mod> depend on the
// subsystem we are part of?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSyst
em()))
```

可以改成以下没有注释的版本：

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

代码原作者可能（不太像）是先写注释再编写代码。不过，作者应该重构代码，如我所做的那样，从而删掉注释。

4.4.9 位置标记

有时，程序员喜欢在源代码中标记某个特别位置。例如，最近我在程序中看到这样一行：

```
// Actions //////////////////////////////////
```

把特定函数趸放在这种标记栏下面，多数时候实属无理。鸡零狗碎，理当删除——特别是尾部那一长串无用的斜杠。

这么说吧。如果标记栏不多，就会显而易见。所以，尽量少用标记栏，只在特别有价值的时候用。如果滥用标记栏，就会沉没在背景噪音中，被忽略掉。

4.4.10 括号后面的注释

有时，程序员会在括号后面放置特殊的注释，如代码清单4-6所示。尽管这对于含有深度嵌套结构的长函数可能有意义，但只会给我们更愿意编写的短小、封装的函数带来混乱。如果你发现自己想标记右括号，其实应该做的是缩短函数。

代码清单4-6 wc.java

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader
(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
        String words[] = line.split("\\W");
        wordCount += words.length;
      } //while



      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try



    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());
    } //catch



  } //main
```

```
}
```

4.4.11　归属与署名

```
/* Added by Rick  */
```

源代码控制系统非常善于记住是谁在何时添加了什么。没必要用那些小小的签名搞脏代码。你也许会认为，这种注释大概有助于他人了解应该和谁讨论这段代码。不过，事实却是注释在那儿放了一年又一年，越来越不准确，越来越和原作者没关系。

重申一下，源代码控制系统是这类信息最好的归属地。

4.4.12　注释掉的代码

直接把代码注释掉是讨厌的做法。别这么干！

```
    InputStreamResponse response = new InputStreamResponse();
    response.setBody(formatter.getResultStream(), formatter.getByteCount());
//   InputStream resultsStream = formatter.getResultStream();
```

```
//   StreamReader reader = new StreamReader(resultsStream);


//   response.setContent(reader.read(formatter.getByteCount()));
```

其他人不敢删除注释掉的代码。他们会想，代码依然放在那儿，一定有其原因，而且这段代码很重要，不能删除。注释掉的代码堆积在一起，就像破酒瓶底的渣滓一般。

看看以下来自Apache公共库的代码：

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;



writeHeader();
writeResolution();
//dataPos = bytePos;



if (writeImageData()) {
  writeEnd();
  this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else{
    this.pngBytes=null;
}
return this.pngBytes;
```

这两行代码为什么要注释掉？它们重要吗？它们搁在那儿，是为了给未来的修改做提示吗？或者，只是某人在多年以前注释掉、懒得清理的过时玩意？

20世纪60年代，曾经有那么一段时间，注释掉的代码可能有用。但我们已经拥有优良的源代码控制系统如此之久，这些系统可以为我们记住不要的代码。我们无需再用注释来标记，删掉即可，它们丢不了。我担保。

4.4.13　HTML注释

源代码注释中的HTML标记是一种厌物，如你在下面代码中所见。编辑器/IDE中的代码本来易于阅读，却因为HTML注释的存在而变得难以卒读。如果注释将由某种工具（例如Javadoc）抽取出来，呈现到网页，那么该是工具而非程序员来负责给注释加上合适的HTML标签。

```
/**
 * Task to run fit tests.
 * This task runs fitnesse tests and publishes the results.
 * <p/>
 * <pre>
 * Usage:
```

```
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quo
t;
 *     classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *            resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 *     suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 *     fitnesseport=&quot;8082&quot;
 *     resultsdir=&quot;${results.dir}&quot;
 *     resultshtmlpage=&quot;fit-results.html&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

4.4.14 非本地信息

假如你一定要写注释，请确保它描述了离它最近的代码。别在本地注释的上下文环境中给出系统级的信息。以下面的Javadoc注释为例，除了那可怕的冗余之外，它还给出了有关默认端口的信息。不过该函数完全没控制到那个所谓默认值。这个注释并未描述该函数，而是在描述系统中远在他方的其他函数。当然，也无法担保在包含那个默认值的代码修改之后，这里的注释也会跟着修改。

```
/**
 * Port on which fitnesse would run. Defaults to <b>8082</b>.
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
```

```
{
  this.fitnessePort = fitnessePort;
}
```

4.4.15 信息过多

别在注释中添加有趣的历史性话题或者无关的细节描述。下列注释来自某个用来测试base64编解码函数的模块。除了RFC文档编号之外，注释中的其他细节信息对于读者完全没有必要。

```
/*
 RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
 Part One: Format of Internet Message Bodies
 section 6.8.  Base64 Content-Transfer-Encoding
 The encoding process represents 24-bit groups of input bits as
 output
 strings of 4 encoded characters. Proceeding from left to right
, a
 24-bit input group is formed by concatenating 3 8-bit input gr
oups.
 These 24 bits are then treated as 4 concatenated 6-bit groups,
 each
 of which is translated into a single digit in the base64 alpha
bet.
 When encoding a bit stream via the base64 encoding, the bit st
ream
 must be presumed to be ordered with the most-significant-bit f
irst.
 That is, the first bit in the stream will be the high-order bi
t in
 the first 8-bit byte, and the eighth bit will be the low-order
 bit in
 the first 8-bit byte, and so on.
 */
```

4.4.16 不明显的联系

注释及其描述的代码之间的联系应该显而易见。如果你不嫌麻烦要写注释，至少让读者能看着注释和代码，并且理解注释所谈何物。

以来自Apache公共库的这段注释为例：

```
/*
 * start with an array that is big enough to hold all the pixel
s
 * (plus filter bytes), and an extra 200 bytes for header info
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) +
 200];
```

过滤器字节是什么？与那个+1有关系吗？或与*3有关？还是与两者皆有关？为什么用200？注释的作用是解释未能自行解释的代码。如果注释本身还需要解释，就太遗憾了。

4.4.17 函数头

短函数不需要太多描述。为只做一件事的短函数选个好名字，通常要比写函数头注释要好。

4.4.18 非公共代码中的Javadoc

虽然Javadoc对于公共API非常有用，但对于不打算作公共用途的代码就令人厌恶了。为系统中的类和函数生成Javadoc页并非总有用，而Javadoc注释额外的形式要求几乎等同于八股文章。

4.4.19 范例

我曾为首个XP Immersion [4] 课程编写了代码清单4-7列出的模块。这个模块几乎是糟糕的代码和坏注释风格的典范。后来Kent Beck当着几十位满腔热情的学生的面重构了这些代码，将其变得令人愉悦。后来，我在拙著*Agile Software Development*，*Principles*，*Patterns*，*and Practices*（中译版《敏捷软件开发：原则、模式与实践》）和*Software Development*（软件开发）杂志的“技艺”专栏的第一篇文章中引用了这个例子。

这个模块最迷人的地方是，有那么一阵，我们中的许多人都认为它“文档做得很好”。如今，我们认为它是一小团乱麻。看看你能发现多少个不同的注释问题吧。

代码清单4-7 GeneratePrimes.java

```
/**
```

```
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * <p>
 * Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 * d. c. 194, Alexandria.  The first man to calculate the
 * circumference of the Earth.  Also known for working on
 * calendars with leap years and ran the library at Alexandria.
 * <p>
 * The algorithm is quite simple.  Given an array of integers
 * starting at 2.  Cross out all multiples of 2.  Find the next
 * uncrossed integer, and cross out all of its multiples.
 * Repeat untilyou have passed the square root of the maximum
 * value.
 *
 * @author Alphonse
 * @version 13 Feb 2002 atp
 */
import java.util.*;
public class GeneratePrimes
{
  /**
   * @param maxValue is the generation limit.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // the only valid case
    {
      // declarations
      int s = maxValue + 1; // size of array
      boolean[] f = new boolean[s];
      int i;
      // initialize array to true.
      for (i = 0; i < s; i++)
        f[i] = true;
      // get rid of known non-primes
      f[0] = f[1] = false;
      // sieve
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // multiple is not prime
        }
```

```
    }

    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
      if (f[i])
        count++; // bump count.
    }

    int[] primes = new int[count];

    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
      if (f[i])            // if prime
        primes[j++] = i;
    }

    return primes;  // return the primes
  }
  else // maxValue < 2
    return new int[0]; // return null array if bad input.
 }
}
```

在代码清单4-8中，你可以看到该模块重构后的版本。注意，注释的使用被明显地限制了。在整个模块中只有两个注释。每个注释都足具说明意义。

代码清单4-8　PrimeGenerator.java（重构后）

```
/**
 * This class Generates prime numbers up to a user specified
 * maximum.  The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
```

```
 * multiples.  Repeat until there are no more multiples
 * in the array.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Every multiple in the array has a prime factor that
    // is less than or equal to the root of the array size,
    // so we don't have to cross out multiples of numbers
    // larger than that root.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
```

```
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
  }

  private static boolean notCrossed(int i)
  {
    return crossedOut[i] == false;
  }

  private static void putUncrossedIntegersIntoResult()
  {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers()
  {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;

    return count;
  }
}
```

很容易说明，第一个注释完全是多余的，因为它读起来非常像是generatePrimes函数自身。不过，我认为这段注释还是省了读者去读具体算法的精力，所以我倾向于留下它。

第二个注释显然很有必要。它解释了平方根作为循环限制的理由。我找不到能说明白这个问题的简单变量名或者其他编程结构。另外，对平方根的使用可能也有点武断。通过限制平方根循环，我是否真节省了许多时间？平方根计算所花的时间会不会比省下的时间还要多？这些都值得考虑。使用平方根作为循环限制，满足了我这种旧式C语言和汇编语言黑客，不过我可不敢说抵得上其他人为理解它而花的时间和精力。

4.5 文献

[KP78]： Kernighan and Plaugher, *The Elements of Programming Style* , 2d. ed., McGraw- Hill, 1978.

---

[1] 原注：[KP78], p. 144。

[2] 译注：意为“运行时间过长”。

[3] 原注：IDE对注释中拼写检查的支持对我们这些看大量代码的人实在是一种妙事。

[4] 译注：Object Mentor公司开办的极限编程深入课程。

第5章　格式



当有人查看底层代码实现时，我们希望他们为其整洁、一致及所感知到的对细节的关注而震惊。我们希望他们高高扬起眉毛，一路看下去。我们希望他们感受到那些为之劳作的专业人士们。但若他们看到的只是一堆像是由酒醉的水手写出的鬼画符，那他们多半会得出结论，认为项目其他任何部分也同样对细节漠不关心。

你应该保持良好的代码格式。你应该选用一套管理代码格式的简单规则，然后贯彻这些规则。如果你在团队中工作，则团队应该一致同意采用一套简单的格式规则，所有成员都要遵从。使用能帮你应用这些格式规则的自动化工具会很有帮助。

5.1 格式的目的

先明确一下，代码格式很重要。代码格式不可忽略，必须严肃对待。代码格式关乎沟通，而沟通是专业开发者的头等大事。

或许你认为“让代码能工作”才是专业开发者的头等大事。然而，我希望本书能让你抛掉那种想法。你今天编写的功能，极有可能在下一版本中被修改，但代码的可读性却会对以后可能发生的修改行为产生深远影响。原始代码修改之后很久，其代码风格和可读性仍会影响到可维护性和扩展性。即便代码已不复存在，你的风格和律条仍存活下来。

那么，哪些代码格式相关方面能帮我们最好地沟通呢？

5.2　垂直格式

从垂直尺寸开始吧。源代码文件该有多大？在Java中，文件尺寸与类尺寸极其相关。讨论类时再说类的尺寸。现在先考虑文件尺寸。

多数Java源代码文件有多大？事实说明，尺寸各各不同，长度殊异，如图5-1所示。

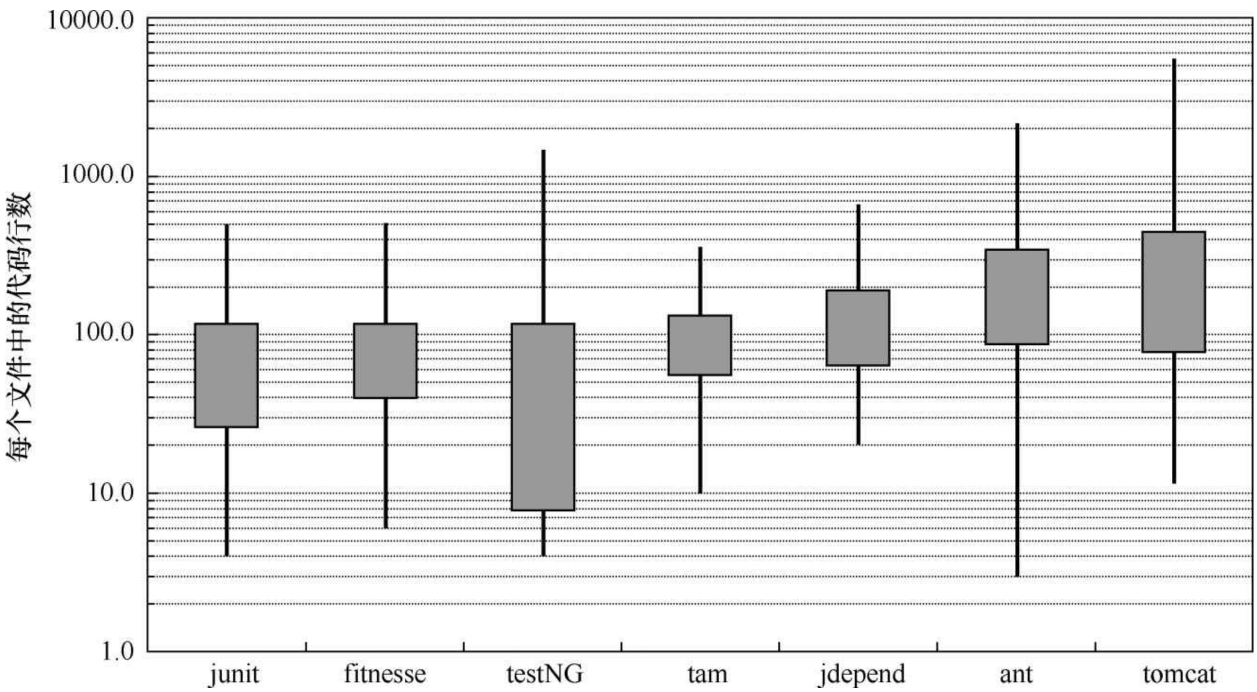


图5-1　以对数标尺显示的文件长度分布（方块高度=sigma）

图5-1中涉及7个不同项目：Junit、FitNesse、testNG、Time and Money、JDepend、Ant和Tomcat。贯穿方块的直线两端显示这些项目中最小

和最大的文件长度。方块表示在平均值以上或以下的大约三分之一文件（一个标准偏差[1]）的长度。方块中间位置就是平均数。所以FitNesse项目的文件平均尺寸是65行，而上面三分之一在40～100行及100行以上之间。FitNesse中最大的文件大约400行，最小是6行。这是个对数标尺，所以较小的垂直位置差异意味着文件绝对尺寸的较大差异。

Junit、FitNesse和Time and Money由相对较小的文件组成。没有一个超过500行，多数都小于200行。Tomcat和Ant则有些文件达到数千行，将近一半文件长于200行。

对我们来说，这意味着什么？意味着有可能用大多数为200行、最长500行的单个文件构造出色的系统（FitNesse总长约50000行）。尽管这并非不可违背的原则，也应该乐于接受。短文件通常比长文件易于理解。

5.2.1　向报纸学习

想想看写得很好的报纸文章。你从上到下阅读。在顶部，你期望有个头条，告诉你故事主题，好让你决定是否要读下去。第一段是整个故事的大纲，给出粗线条概述，但隐藏了故事细节。接着读下去，细节渐次增加，直至你了解所有的日期、名

字、引语、说法及其他细节。

源文件也要像报纸文章那样。名称应当简单且一目了然。名称本身应该足够告诉我们是否在正确的模块中。源文件最顶部应该给出高层次概念和算法。细节应该往下渐次展开，直至找到源文件中最底层的函数和细节。

报纸由许多篇文章组成；多数短小精悍。有些稍微长点儿。很少有占满一整页的。这样做，报纸才可用。假若一份报纸只登载一篇长故事，其中充斥毫无组织的事实、日期、名字等，没人会去读它。

5.2.2　概念间垂直方向上的区隔

几乎所有的代码都是从上往下读，从左往右读。每行展现一个表达式或一个子句，每组代码行展示一条完整的思路。这些思路用空白行区隔开来。

以代码清单5-1为例。在封包声明、导入声明和每个函数之间，都有空白行隔开。这条极其简单的规则极大地影响到代码的视觉外观。每个空白行都是一条线索，标识出新的独立概念。往下读代码时，你的目光总会停留于空白行之后那一行。

代码清单5-1　BoldWidget.java

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+
?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exc
eption {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

如代码清单5-2所示，抽掉这些空白行， 代码可读性减弱了不少。

代码清单5-2　BoldWidget.java

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+
```

```
?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exc
eption {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

在你不特意注视时，后果就更严重了。在第一个例子中，代码组会跳到你眼中，而第二个例子就像一堆乱麻。两段代码的区别，展示了垂直方向上区隔的作用。

5.2.3　垂直方向上的靠近

如果说空白行隔开了概念，靠近的代码行则暗示了它们之间的紧密关系。所以，紧密相关的代码应该互相靠近。注意代码清单5-3中的注释是如何割断两个实体变量间的联系的。

代码清单5-3

```
public class ReporterConfig {
  /**
   * The class name of the reporter listener
   */
```

```
  private String m_className;

  /**
   * The properties of the reporter listener
   */
  private List<Property> m_properties = new ArrayList<Property>(
);

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

代码清单5-4更易于阅读。它刚好"一览无遗"，至少对我来说是这样。我一眼就能看到，这是个有两个变量和一个方法的类。看上面的代码时，我不得不更多地移动头部和眼球，才能获得相同的理解度。

代码清单5-4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>(
);

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

5.2.4 垂直距离

你是否曾经在某个类中摸索，从一个函数跳到

另一个函数，上下求索，想要弄清楚这些函数如何操作、如何互相相关，最后却被搞糊涂了？你是否曾经苦苦追索某个变量或函数的继承链条？这让人沮丧，因为你想要理解系统做什么，但却花时间和精力在找到和记住那些代码碎片在哪里。

关系密切的概念应该互相靠近[G10]。显然，这条规则并不适用于分布在不同文件中的概念。除非有很好的理由，否则就不要把关系密切的概念放到不同的文件中。实际上，这也是避免使用protected变量的理由之一。

对于那些关系密切、放置于同一源文件中的概念，它们之间的区隔应该成为对相互的易懂度有多重要的衡量标准。应避免迫使读者在源文件和类中跳来跳去。

变量声明。变量声明应尽可能靠近其使用位置。因为函数很短，本地变量应该在函数的顶部出现，就像Junit4.3.1中这个稍长的函数中那样。

```
private static void readPreferences() {
 InputStream is= null;



  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
```

```
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

循环中的控制变量应该总是在循环语句中声明，如下列来自同一项目的绝妙小函数所示。

```
public int countTestCases() {
  int count= 0;
  for (Test each


 : tests)
     count += each.countTestCases();
  return count;
}
```

偶尔，在较长的函数中，变量也可能在某个代码块顶部，或在循环之前声明。你可以在以下摘自TestNG中一个长函数的代码片段中找到类似的变量。

```
...
for (XmlTest test : m_suite.getTests()) {
       TestRunner tr = m_runnerFactory.newTestRunner(this, test);
```

```
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }

      for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
       afterSuiteMethods.put(m.getMethod(), m);
      }
    }
...
```

实体变量 应该在类的顶部声明。这应该不会增加变量的垂直距离，因为在设计良好的类中，它们如果不是被该类的所有方法也是被大多数方法所用。

关于实体变量应该放在哪里，争论不断。在C++中，通常会采用所谓“剪刀原则 ”（scissors rule），所有实体变量都放在底部。而在Java中，惯例是放在类的顶部。没理由去遵循其他惯例。重点是在谁都知道的地方声明实体变量。大家都应该知道在哪儿能看到这些声明。

例如JUnit 4.3.1中的这个奇怪情形。我极力删

减了这个类，好说明问题。如果你看到代码清单大致一半的位置，会看到在那里声明了两个实体变量。如果放在更好的位置，它们就会更明显。而现在，读代码者只能在无意中看到这些声明（就像我一样）。

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theCla
ss,
                                String name) {
    ...
  }

  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
    ...
  }

  public static Test warning(final String message) {
    ...
  }

  private static String exceptionToString(Throwable t) {
    ...
  }

  private String fName;



  private Vector<Test> fTests= new Vector<Test>(10);



  public TestSuite() {
  }

```

```
  public TestSuite(final Class<? extends TestCase> theClass) {
    ...
  }

  public TestSuite(Class<? extends TestCase>  theClass, String n
ame) {
    ...
  }
  ... ... ... ... ...
}
```

相关函数 。若某个函数调用了另外一个，就应该把它们放到一起，而且调用者应该尽可能放在被调用者上面。这样，程序就有个自然的顺序。若坚定地遵循这条约定，读者将能够确信函数声明总会在其调用后很快出现。以源自FitNesse的代码清单5-5为例。注意顶部的函数是如何调用其下的函数，而这些被调用的函数又是如何调用更下面的函数的。这样就能轻易找到被调用的函数，极大地增强了整个模块的可读性。

代码清单5-5　WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request
request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage")
```

```
;
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String de
faultPageName)
 {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;
    return pageName;
  }

  protected void loadPage(String resource, FitNesseContext conte
xt)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Req
uest request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request
);
  }

  private SimpleResponse makePageResponse(FitNesseContext contex
t)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
```

```
}
...
```

说句题外话，以上代码片段也是把常量保持在恰当级别的好例子[G35]。FrontPage常量可以埋在getPageNameOrDefault函数中，但那样就会把一个众人皆知的常量埋藏到位于不太合适的底层函数中。更好的做法是把它放在易于找到的位置，然后再传递到真实使用的位置。



概念相关 。概念相关的代码应该放到一起。相关性越强，彼此之间的距离就该越短。

如上所述，相关性应建立在直接依赖的基础上，如函数间调用，或函数使用某个变量。但也有

其他相关性的可能。相关性可能来自于执行相似操作的一组函数。请看以下来自Junit 4.3.1的代码片段：

```
public class Assert {
  static public void assertTrue(String message, boolean conditio
n) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean conditi
on) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
  ...
```

这些函数有着极强的概念相关性，因为他们拥有共同的命名模式，执行同一基础任务的不同变种。互相调用是第二位的。即便没有互相调用，也应该放在一起。

5.2.5　垂直顺序

一般而言，我们想自上向下展示函数调用依赖

顺序。也就是说，被调用的函数应该放在执行调用的函数下面 [2] 。这样就建立了一种自顶向下贯穿源代码模块的良好信息流。

像报纸文章一般，我们指望最重要的概念先出来，指望以包括最少细节的方式表述它们。我们指望底层细节最后出来。这样，我们就能扫过源代码文件，自最前面的几个函数获知要旨，而不至于沉溺到细节中。代码清单5-5就是如此组织的。或许，更好的例子是代码清单15-5，及代码清单3-7。

5.3　横向格式

一行代码应该有多宽？要回答这个问题，来看看典型的程序中代码行的宽度。我们再一次检验7个不同项目。图5-2展示了这7个项目的代码行宽度分布情况。其中展现的规律性令人印象深刻，45个字符左右的宽度分布尤为如此。其实，20～60的每个尺寸，都代表全部代码行数的1%。也就是总共40%！或许其余30%的代码行短于10个字符。记住，这是个对数标尺，所以图中长于80个字符部分的线性下降在实际情况中会极其可观。程序员们显然更喜爱短代码行。

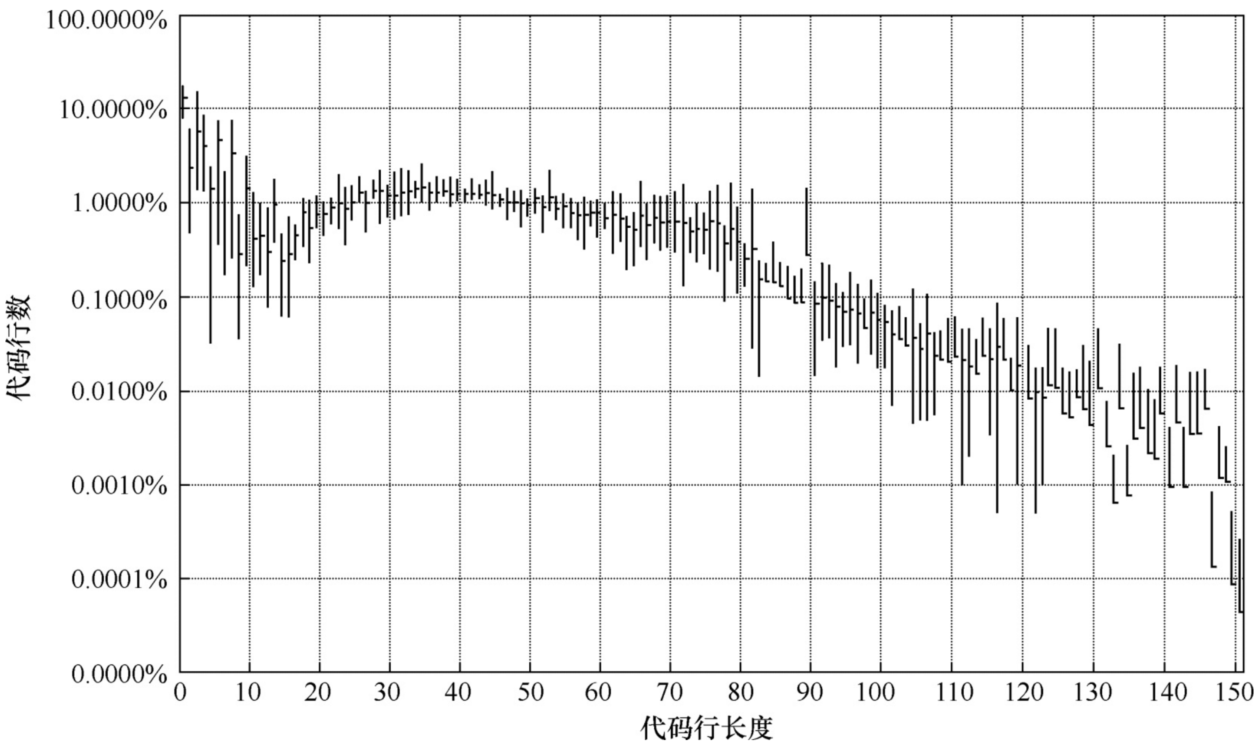


图5-2　Java程序代码行长度分布

这说明，应该尽力保持代码行短小。死守80个字符的上限有点僵化，而且我也并不反对代码行长度达到100个字符或120个字符。再多的话，大抵就是肆意妄为了。

我一向遵循无需拖动滚动条到右边的原则。但近年来显示器越来越宽，而年轻程序员又能将显示字符缩小到如此程度，屏幕上甚至能容纳200个字符的宽度。别那么做。我个人的上限是120个字符。

5.3.1　水平方向上的区隔与靠近

我们使用空格字符将彼此紧密相关的事物连接到一起，也用空格字符把相关性较弱的事物分隔开。请看以下函数：

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

我在赋值操作符周围加上空格字符，以此达到强调目的。赋值语句有两个确定而重要的要素：左边和右边。空格字符加强了分隔效果。

另一方面，我不在函数名和左圆括号之间加空格。这是因为函数与其参数密切相关，如果隔开，就会显得互无关系。我把函数调用括号中的参数一一隔开，强调逗号，表示参数是互相分离的。

空格字符的另一种用法是强调其前面的运算符。

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
```

```
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c
) {
    return b*b - 4*a*c;
  }
}
```

看看这些等式读起来多舒服。乘法因子之间没加空格，因为它们具有较高优先级。加减法运算项之间用空格隔开，因为加法和减法优先级较低。

不幸的是，多数代码格式化工具都会漠视运算符优先级，从头到尾采用同样的空格方式。在重新格式化代码后，以上这些微妙的空格用法就消失殆尽了。

5.3.2　水平对齐

当我还是个汇编语言程序员时[3]，使用水平对齐来强调某些程序结构。开始用C、C++编码，最终转向Java后，我继续尽力对齐一组声明中的变量名，或一组赋值语句中的右值。我的代码看起来大概是这样：

```
public class FitNesseExpediter implements ResponseSender
{
  private    Socket            socket;
```

```
  private    InputStream      input;
  private   OutputStream     output;
  private    Request          request;
  private    Response         response;
  private    FitNesseContext  context;
  protected long             requestParsingTimeLimit;
  private    long             requestProgress;
  private    long             requestParsingDeadline;
  private    boolean          hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Excepti
on
{
    this.context =        context;
    socket =              s;
    input =               s.getInputStream();
    output =              s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

我发现这种对齐方式没什么用。对齐，像是在强调不重要的东西，把我的目光从真正的意义上拉开。例如，在上面的声明列表中，你会从上到下阅读变量名，而忽视了它们的类型。同样，在赋值语句代码清单中，你也会从上到下阅读右值，而对赋值运算符视而不见。更麻烦的是，代码自动格式化工具通常会把这类对齐消除掉。

所以，我最终放弃了这种做法。如今，我更喜欢用不对齐的声明和赋值，如下所示，因为它们指出了重点。如果有较长的列表需要做对齐处理，那问题就是在列表的长度上 而不是对齐上。下例

FitNesseExpediter类中声明列表的长度说明该类应该被拆分了。

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;

  public FitNesseExpediter(Socket s, FitNesseContext context) th
rows Exception
  {
    this.context = context;
    socket = s;
    input = s.getInputStream();
    output = s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

5.3.3 缩进

源文件是一种继承结构，而不是一种大纲结构。其中的信息涉及整个文件、文件中每个类、类中的方法、方法中的代码块，也涉及代码块中的代码块。这种继承结构中的每一层级都圈出一个范围，名称可以在其中声明，而声明和执行语句也可以在其中解释。

要让这种范围式继承结构可见，我们依源代码行在继承结构中的位置对源代码行做缩进处理。在文件顶层的语句，例如大多数的类声明，根本不缩进。类中的方法相对该类缩进一个层级。方法的实现相对方法声明缩进一个层级。代码块的实现相对于其容器代码块缩进一个层级，以此类推。

程序员相当依赖这种缩进模式。他们从代码行左边查看自己在什么范围中工作。这让他们能快速跳过与当前关注的情形无关的范围，例如if或while语句的实现之类。他们的眼光扫过左边，查找新的方法声明、新变量，甚至新类。没有缩进的话，程序就会变得无法阅读。

试看以下在语法和语义上等价的两个程序：

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----

public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
  public FitNesseServer(FitNesseContext context) {
```

```
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, contex
t);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

你能很快地洞悉有缩进的那个文件的结构。你几乎能立即就辨别出那些变量、构造器、存取器和方法。只需要几秒钟就能了解这是一个套接字的简单前端，其中包括了超时设定。而未缩进的版本则不经过一番折腾就无法明白。

违反缩进规则 。有时，会忍不住想要在短小的if语句、while循环或小函数中违反缩进规则。一旦这么做了，我多数时候还是会回头加上缩进。这样就避免了出现以下这种范围层级坍塌到一行的情况：

```
public class CommentWidget extends TextWidget
{
```

```
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|
\r)?";

  public CommentWidget(ParentWidget parent, String text){super(p
arent, text);}
  public String render() throws Exception {return ""; }
}
```

我更喜欢扩展和缩进范围，就像这样：

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|
\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }
  public String render() throws Exception {
    return "";
  }
}
```

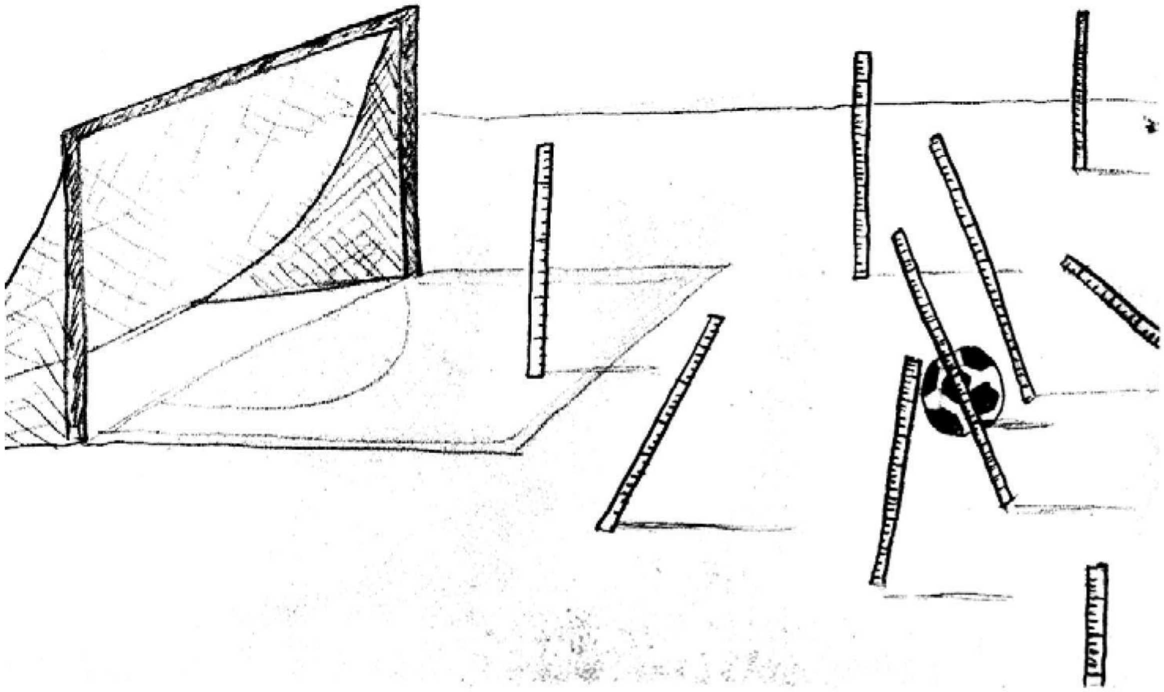
5.3.4　空范围

有时，while或for语句的语句体为空，如下所示。我不喜欢这种结构，尽量不使用。如果无法避免，就确保空范围体的缩进，用括号包围起来。我无法告诉你，我曾经多少次被静静安坐在与while循环语句同一行末尾的分号所欺骗。除非你把那个分号放到另一行再加以缩进，否则就很难看到它。

```
while (dis.read(buf, 0, readBufferSize) != -1)
;
```

5.4 团队规则

每个程序员都有自己喜欢的格式规则，但如果在一个团队中工作，就是团队说了算[4]。



一组开发者应当认同一种格式风格，每个成员都应该采用那种风格。我们想要让软件拥有一以贯之的风格。我们不想让它显得是由一大票意见相左的个人所写成。

2002年启动FitNesse项目时，我和开发团队一起制订了一套编码风格。这只花了我们10分钟时间。我们决定了在什么地方放置括号，缩进几个字符，如何命名类、变量和方法，如此等等。然后，我们把这些规则编写进IDE的代码格式功能，接着就一直沿用。这些规则并非全是我喜爱的；但它们是团队决定了的规则。作为团队一员，在为FitNesse项目编写代码时，我遵循这些规则。

记住，好的软件系统是由一系列读起来不错的代码文件组成的。它们需要拥有一致和顺畅的风格。读者要能确信，他们在一个源文件中看到的格式风格在其他文件中也是同样的用法。绝对不要用各种不同的风格来编写源代码，这样会增加其复杂度。

5.5　鲍勃大叔的格式规则

我个人使用的规则相当简单，如代码清单5-6所示。可以把这段代码看作是展示如何把代码写成最好的编码标准文档的范例。

代码清单5-6　CodeAnalyzer.java

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;

  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }

  public static List<File> findJavaFiles(File parentDirectory) {
    List<File> files = new ArrayList<File>();
    findJavaFiles(parentDirectory, files);
    return files;
  }

  private static void findJavaFiles(File parentDirectory, List<F
ile> files) {
    for (File file : parentDirectory.listFiles()) {
      if (file.getName().endsWith(".java"))
        files.add(file);
      else if (file.isDirectory())
        findJavaFiles(file, files);
    }
  }

  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFi
```

```
le));
    String line;
    while ((line = br.readLine()) != null)
      measureLine(line);
  }

  private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }

  private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
      maxLineWidth = lineSize;
      widestLineNumber = lineCount;
    }
  }

  public int getLineCount() {
    return lineCount;
  }

  public int getMaxLineWidth() {
    return maxLineWidth;
  }

  public int getWidestLineNumber() {
    return widestLineNumber;
  }

  public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
  }

  public double getMeanLineWidth() {
    return (double) totalChars / lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
```

```
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount / 2)
        return width;
    }
    throw new Error("Cannot  get  here");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
  }
  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

---

[1]　原注：方块显示平均数的sigma/2以上及以下长度。没错，我知道文件长度分布不太寻常，所以标准偏差也并非那么精确。不过在此并不寻求精确，只是找个感觉罢了。

[2]　原注：Pascal、C和C++等语言中完全不同，在这些语言中，函数应该在被调用之前定义，至少是声明。

[3]　原注：开什么玩笑！到现在我仍是个汇编语言程序员。把男孩从铁旁边赶走容易，从男孩身边把铁拿走可难！

[4] 译注：团队规则，原文team rules。单词rule在这里有两个意思，一个是名词“规则”，一个是动词“管辖”，所以本节标题玩了个文字游戏。中文不易翻出，故采取意译加注。

第6章　对象和数据结构



将变量设置为私有（private）有一个理由：我们不想其他人依赖这些变量。我们还想在心血来潮时能自由修改其类型或实现。那么，为什么还是有

那么多程序员给对象自动添加赋值器和取值器，将私有变量公之于众、如同它们根本就是公共变量一般呢？

6.1　数据抽象

看看代码清单6-1和代码清单6-2之间的区别。每段代码都表示笛卡儿平面上的一个点。不过，其中之一曝露了其实现，而另一个则完全隐藏了其实现。

代码清单6-1　具象点

```
public class Point {
  public double x;
  public double y;
}
```

代码清单6-2　抽象点

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

代码清单6-2的漂亮之处在于，你不知道该实现会是在矩形坐标系中还是在极坐标系中。可能两个都不是！然而，该接口还是明白无误地呈现了一

种数据结构。

不过它呈现的还不止是一个数据结构。那些方法固定了一套存取策略。你可以单独读取某个坐标，但必须通过一次原子操作设定所有坐标。

而代码清单6-1则非常清楚地是在矩形坐标系中实现，并要求我们单个操作那些坐标。这就曝露了实现。实际上，即便变量都是私有，而且我们也通过变量取值器和赋值器使用变量，其实现仍然曝露了。

隐藏实现并非只是在变量之间放上一个函数层那么简单。隐藏实现关乎抽象！类并不简单地用取值器和赋值器将其变量推向外间，而是曝露抽象接口，以便用户无需了解数据的实现就能操作数据本体 。

看看代码清单6-3和代码清单6-4。前者使用具象手段与机动车的燃料层通信，而后者则采用百分比抽象。你能确定前者里面都是些变量存取器，而却无法得知后者中的数据形态。

代码清单6-3　具象机动车

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
```

```
}
```

代码清单6-4　抽象机动车

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

以上两段代码以后者为佳。我们不愿曝露数据细节，更愿意以抽象形态表述数据。这并不只是用接口和/或赋值器、取值器就万事大吉。要以最好的方式呈现某个对象包含的数据，需要做严肃的思考。傻乐着乱加取值器和赋值器，是最坏的选择。

6.2　数据、对象的反对称性

这两个例子展示了对象与数据结构之间的差异。对象把数据隐藏于抽象之后，曝露操作数据的函数。数据结构曝露其数据，没有提供有意义的函数。回过头再读一遍。留意这两种定义的本质。它们是对立的。这种差异貌似微小，但却有深远的含义。

例如，代码清单6-5中的过程式代码形状范例。Geometry类操作三个形状类。形状类都是简单的数据结构，没有任何行为。所有行为都在Geometry类中。

代码清单6-5　过程式形状代码

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```

```
public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

面向对象程序员可能会对此嗤之以鼻，抱怨说这是过程式代码——他们大概是对的，不过这种嘲笑并不完全正确。想想看，如果给Geometry类添加一个primeter( )函数会怎样。那些形状类根本不会因此而受影响！另一方面，如果添加一个新形状，就得修改Geometry中的所有函数来处理它。再读一遍代码。注意，这两种情形也是直接对立的。

现在来看看代码清单6-6中的面向对象方案。这里，area( )方法是多态的。不需要有Geometry类。所以，如果添加一个新形状，现有的函数 一个也不会受到影响，而当添加新函数时所有的形状

都得做修改[1]！

代码清单6-6　多态式形状

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}

public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

我们再次看到这两种定义的本质；它们是截然对立的。这说明了对象与数据结构之间的二分原理：

过程式代码（使用数据结构的代码）便于在不改动既有数据结构的前提下添加新函数。面向对象代码便于在不改动既有函数的前提下添加新类。

反过来讲也说得通：

过程式代码难以添加新数据结构，因为必须修改所有函数。面向对象代码难以添加新函数，因为必须修改所有类。

所以，对于面向对象较难的事，对于过程式代码却较容易，反之亦然！

在任何一个复杂系统中，都会有需要添加新数据类型而不是新函数的时候。这时，对象和面向对象就比较适合。另一方面，也会有想要添加新函数而不是数据类型的时候。在这种情况下，过程式代码和数据结构更合适。

老练的程序员知道，一切都是对象只是一个传说。有时候你真的想要在简单数据结构上做一些过程式的操作。

6.3　得墨忒耳律

著名的得墨忒耳律（The Law of Demeter）[2]认为，模块不应了解它所操作对象 的内部情形。如上节所见，对象隐藏数据，曝露操作。这意味着对象不应通过存取器曝露其内部结构，因为这样更像是曝露而非隐藏其内部结构。

更准确地说，得墨忒耳律认为，类C的方法f只应该调用以下对象的方法：

- C
- 由f创建的对象；
- 作为参数传递给f的对象；
- 由C的实体变量持有的对象。

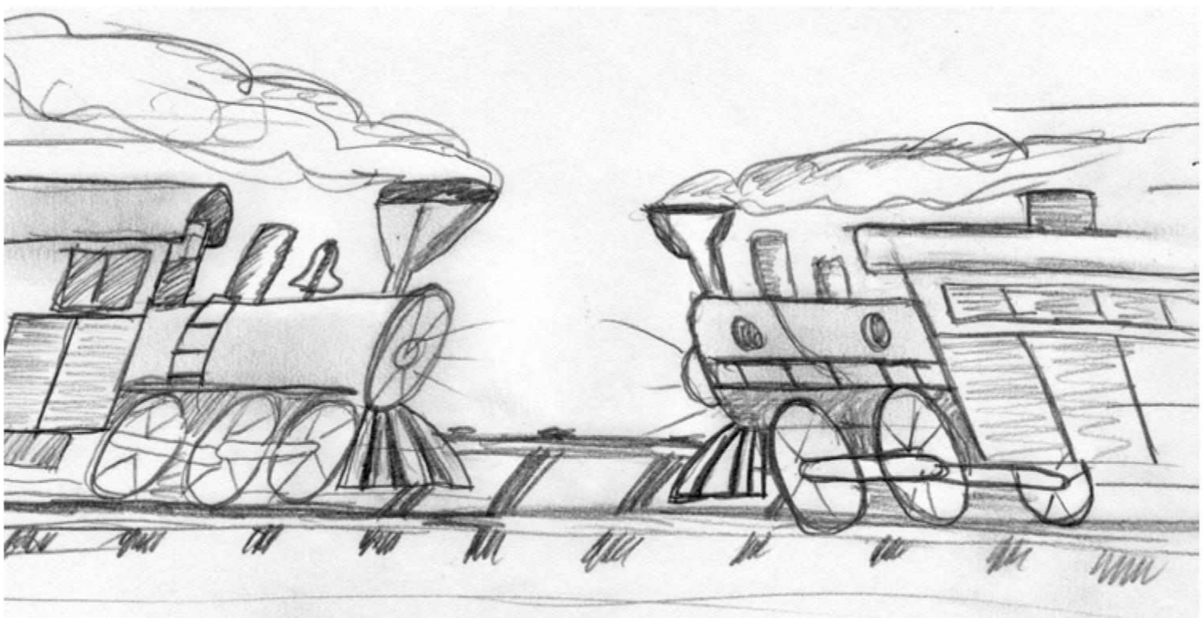
方法不应 调用由任何函数返回的对象的方法。换言之，只跟朋友谈话，不与陌生人谈话。

下列代码[3]违反了得墨忒耳律（除了违反其他规则之外），因为它调用了getOptions( )返回值的getScratchDir( )函数，又调用了getScratchDir( )返回值的getAbsolutePath( )方法。

```
final String outputDir = ctxt.getOptions().getScratchDir().getA
```

```
bsolutePath();
```

6.3.1 火车失事



这类代码常被称作火车失事，因为它看起来就像是一列火车。这类连串的调用通常被认为是肮脏的风格，应该避免[G36]。最好做类似如下的切分：

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

上列代码是否违反了得墨忒耳律呢？当然，模

块知道ctxt对象包含有多个选项，每个选项中都有一个临时目录，而每个临时目录都有一个绝对路径。对于一个函数，这些知识真够丰富的。调用函数懂得如何在一大堆不同对象间浏览。

这些代码是否违反得墨忒耳律，取决于ctxt、Options和ScratchDir是对象还是数据结构。如果是对象，则它们的内部结构应当隐藏而不曝露，而有关其内部细节的知识就明显违反了得墨忒耳律。如果ctxt、Options和ScratchDir只是数据结构，没有任何行为，则它们自然会曝露其内部结构，得墨忒耳律也就不适用了。

属性访问器函数的使用把问题搞复杂了。如果像下面这样写代码，我们大概就不会提及对得墨忒耳律的违反。

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

如果数据结构只简单地拥有公共变量，没有函数，而对象则拥有私有变量和公共函数，这个问题就不那么混淆。然而，有些框架和标准甚至要求最简单的数据结构都要有访问器和改值器。

6.3.2　混杂

这种混淆有时会不幸导致混合结构，一半是对象，一半是数据结构。这种结构拥有执行操作的函数，也有公共变量或公共访问器及改值器。无论出于怎样的初衷，公共访问器及改值器都把私有变量公开化，诱导外部函数以过程式程序使用数据结构的方式使用这些变量[4]。

此类混杂增加了添加新函数的难度，也增加了添加新数据结构的难度，两面不讨好。应避免创造这种结构。它们的出现，展示了一种乱七八糟的设计，其作者不确定——或者更糟糕，完全无视——他们是否需要函数或类型的保护。

6.3.3 隐藏结构

假使ctxt、Options和ScratchDir是拥有真实行为的对象又怎样呢？由于对象应隐藏其内部结构，我们就不该能够看到内部结构。这样一来，如何才能取得临时目录的绝对路径呢？

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

或者

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

第一种方案可能导致ctxt对象中方法的曝露。第二种方案是在假设getScratchDirectoryOption()返回一个数据结构而非对象。两种方案感觉都不好。

如果ctxt是个对象，就应该要求它做点什么，不该要求它给出内部情形。那我们为何还要得到临时目录的绝对路径呢？我们要它做什么？来看看同一模块（许多行之后）的这段代码：

```
String outFile = outputDir + "/" + className.replace('.', '/')
+ ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

这种不同层级细节的混杂（[G34][G36]）有点麻烦。句点、斜杠、文件扩展名和File对象不该如此随便地混杂到一起。不过，撇开这些毛病，我们发现，取得临时目录绝对路径的初衷是为了创建指定名称的临时文件。

所以，直接让ctxt对象来做这事如何？

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFi
leName);
```

这下看起来像是个对象做的事了！ctxt隐藏了其内部结构，防止当前函数因浏览它不该知道的对象而违反得墨忒耳律。

6.4　数据传送对象

最为精练的数据结构，是一个只有公共变量、没有函数的类。这种数据结构有时被称为数据传送对象，或DTO（Data Transfer Objects）。DTO是非常有用的结构，尤其是在与数据库通信、或解析套接字传递的消息之类场景中。在应用程序代码里一系列将原始数据转换为数据库的翻译过程中，它们往往是排头兵。

更常见的是如代码清单6-7所示的“bean”结构。豆结构拥有由赋值器和取值器操作的私有变量。对豆结构的半封装会让某些OO纯化论者感觉舒服些，不过通常没有其他好处。

代码清单6-7　address.java

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
```

```
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
}

  public String getZip() {
    return zip;
  }
}
```

Active Record

Active Record是一种特殊的DTO形式。它们是拥有公共（或可豆式访问的）变量的数据结构，但通常也会拥有类似save和find这样的可浏览方法。Active Record一般是对数据库表或其他数据源的直接翻译。

我们不幸经常发现开发者往这类数据结构中塞进业务规则方法，把这类数据结构当成对象来用。这是不智的行为，因为它导致了数据结构和对象的

混杂体。

当然，解决方案就是把Active Record当做数据结构，并创建包含业务规则、隐藏内部数据（可能就是Active Record的实体）的独立对象。

6.5　小结

对象曝露行为，隐藏数据。便于添加新对象类型而无需修改既有行为，同时也难以在既有对象中添加新行为。数据结构曝露数据，没有明显的行为。便于向既有数据结构添加新行为，同时也难以向既有函数添加新数据结构。

在任何系统中，我们有时会希望能够灵活地添加新数据类型，所以更喜欢在这部分使用对象。另外一些时候，我们希望能灵活地添加新行为，这时我们更喜欢使用数据类型和过程。优秀的软件开发者不带成见地了解这种情形，并依据手边工作的性质选择其中一种手段。

6.6　文献

[Refactoring]：　*Refactoring: Improving the Design of Existing Code* , Martin Fowler et al., Addison-Wesley, 1999.

---

[1]　原注：经验丰富的面向对象设计人员都知道一些方法，例如，VISITOR模式，或双向分派。但这些技法也有成本，而且通常返回一种过程式程序的结构。

[2]　原注：http://en.wikipedia.org/wiki/Law_of_Demeter 。

[3]　原注：来自Apache框架中某处。

[4]　原注：在*Refactoring*：*Improving the Design of Existing Code* （中译版《重构改善既有代码的设计（中文版）》）一书中，有时把这种情况称作特性依恋（Feature Envy）。

第7章　错误处理

Michael Feathers



在一本有关整洁代码的书中，居然有讨论错误处理的章节，看起来有些突兀。错误处理只不过是编程时必须要做的事之一。输入可能出现异常，设备可能失效。简言之，可能会出错，当错误发生时，程序员就有责任确保代码照常工作。

然而，应该弄清楚错误处理与整洁代码的关系。许多程序完全由错误处理所占据。所谓占据，并不是说错误处理就是全部。我的意思是几乎无法看明白代码所做的事，因为到处都是凌乱的错误处

理代码。错误处理很重要，但如果它搞乱了代码逻辑，就是错误的做法 。

在本章中，我将概要列出编写既整洁又强固的代码——雅致地处理错误代码的一些技巧和思路。

7.1　使用异常而非返回码

在很久以前，许多语言都不支持异常。这些语言处理和汇报错误的手段都有限。你要么设置一个错误标识，要么返回给调用者检查的错误码。代码清单7-1中的代码展示了这些手段。

代码清单7-1　DeviceController.java

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
      // Save the device status to the record field
      retrieveDeviceRecord(handle);
      // If not suspended, shut down
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended.  Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

这类手段的问题在于，它们搞乱了调用者代

码。调用者必须在调用之后即刻检查错误。不幸的是，这个步骤很容易被遗忘。所以，遇到错误时，最好抛出一个异常。调用代码很整洁，其逻辑不会被错误处理搞乱。

代码清单7-2展示了在方法中遇到错误时抛出异常的情形。

代码清单7-2　DeviceController.java（采用异常处理）

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }
private void tryToShutDown() throws DeviceShutDownError {
  DeviceHandle handle = getHandle(DEV1);
  DeviceRecord record = retrieveDeviceRecord(handle);

pauseDevice(handle);
clearDeviceWorkQueue(handle);
closeDevice(handle);
}

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.to
String());
    ...
  }

  ...
}
```

注意这段代码整洁了很多。这不仅关乎美观。这段代码更好，因为之前纠结的两个元素设备关闭算法和错误处理现在被隔离了。你可以查看其中任一元素，分别理解它。

7.2 先写Try-Catch-Finally语句

异常的妙处之一是，它们在程序中定义了一个范围 。执行try-catch-finally语句中try部分的代码时，你是在表明可随时取消执行，并在catch语句中接续。

在某种意义上，try代码块就像是事务。catch代码块将程序维持在一种持续状态，无论try代码块中发生了什么均如此。所以，在编写可能抛出异常的代码时，最好先写出try-catch-finally语句。这能帮你定义代码的用户应该期待什么，无论try代码块中执行的代码出什么错都一样。

来看个例子。我们要编写访问某个文件并读出一些序列化对象的代码。

先写一个单元测试，其中显示当文件不存在时将得到一个异常：

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

该测试驱动我们创建以下占位代码：

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // dummy return until we have a real implementation
  return new ArrayList<RecordedGrip>();
}
```

测试失败了，因为以上代码并未抛出异常。下一步，修改实现代码，尝试访问非法文件。该操作抛出一个异常：

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

这次测试通过了，因为我们捕获了异常。此时，我们可以重构了。我们可以缩小异常类型的范围，使之符合FileInputStream构造器真正抛出的异常，即FileNotFoundException：

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
```

```
  }
  return new ArrayList<RecordedGrip>();
}
```

如此一来，我们就用try-catch结构定义了一个范围，可以继续用测试驱动（TDD）方法构建剩余的代码逻辑。这些代码逻辑将在FileInputStream和close之间添加，装作一切正常的样子。

尝试编写强行抛出异常的测试，再往处理器中添加行为，使之满足测试要求。结果就是你要先构造try代码块的事务范围，而且也会帮助你维护好该范围的事务特征。

7.3　使用不可控异常

辩论业已结束。多年来，Java程序员们一直在争论可控异常（checked exception）的利与弊。Java的第一个版本中引入可控异常时，看似一个极好的点子。每个方法的签名都列出它可能传递给调用者的异常。而且，这些异常就是方法类型的一部分。如果签名与代码实际所做之事不符，代码在字面上就无法编译。

那时，我们认为可控异常是个绝妙的主意；而且，它也有所 裨益。然而，现在已经很清楚，对于强固软件的生产，它并非必需。C#不支持可控异常。尽管做过勇敢的尝试，C++最后也不支持可控异常。Python和Ruby同样如此。不过，用这些语言也有可能写出强固的软件。我们得决定——的确如此——可控异常是否值回票价。

代价是什么？可控异常的代价就是违反开放/闭合原则 [1] 。如果你在方法中抛出可控异常，而catch语句在三个层级之上，你就得在**catch**语句和抛出异常处之间的每个方法签名中声明该异常 。这意味着对软件中较低层级的修改，都将波及较高层级的签名。修改好的模块必须重新构建、发布，

即便它们自身所关注的任何东西都没改动过。

以某个大型系统的调用层级为例。顶端函数调用它们之下的函数，逐级向下。假设某个位于最底层级的函数被修改为抛出一个异常。如果该异常是可控的，则函数签名就要添加throw子句。这意味着每个调用该函数的函数都要修改，捕获新异常，或在其签名中添加合适的throw子句。以此类推。最终得到的就是一个从软件最底端贯穿到最高端的修改链！封装被打破了，因为在抛出路径中的每个函数都要去了解下一层级的异常细节。既然异常旨在让你能在较远处处理错误，可控异常以这种方式破坏封装简直就是一种耻辱。

如果你在编写一套关键代码库，则可控异常有时也会有用：你必须捕获异常。但对于一般的应用开发，其依赖成本要高于收益。

7.4　给出异常发生的环境说明

你抛出的每个异常，都应当提供足够的环境说明，以便判断错误的来源和处所。在Java中，你可以从任何异常里得到堆栈踪迹（stack trace）；然而，堆栈踪迹却无法告诉你该失败操作的初衷。

应创建信息充分的错误消息，并和异常一起传递出去。在消息中，包括失败的操作和失败类型。如果你的应用程序有日志系统，传递足够的信息给catch块，并记录下来。

7.5 依调用者需要定义异常类

对错误分类有很多方式。可以依其来源分类：是来自组件还是其他地方？或依其类型分类：是设备错误、网络错误还是编程错误？不过，当我们在应用程序中定义异常类时，最重要的考虑应该是它们如何被捕获 。

来看一个不太好的异常分类例子。下面的try-catch-finally语句是对某个第三方代码库的调用。它覆盖了该调用可能抛出的所有异常：

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

语句包含了一大堆重复代码，这并不出奇。在

大多数异常处理中，不管真实原因如何，我们总是做相对标准的处理。我们得记录错误，确保能继续工作。

在本例中，既然知道我们所做的事不外如此，就可以通过打包调用API、确保它返回通用异常类型，从而简化代码。

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

LocalPort类就是个简单的打包类，捕获并翻译由ACMEPort类抛出的异常：

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
```

```
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

类似我们为ACMEPort定义的这种打包类非常有用。实际上，将第三方API打包是个良好的实践手段。当你打包一个第三方API，你就降低了对它的依赖：未来你可以不太痛苦地改用其他代码库。在你测试自己的代码时，打包也有助于模拟第三方调用。

打包的好处还在于你不必绑死在某个特定厂商的API设计上。你可以定义自己感觉舒服的API。在上例中，我们为port设备错误定义了一个异常类型，然后发现这样能写出更整洁的代码。

对于代码的某个特定区域，单一异常类通常可行。伴随异常发送出来的信息能够区分不同错误。如果你想要捕获某个异常，并且放过其他异常，就使用不同的异常类。

7.6　定义常规流程

如果你遵循前文提及的建议，在业务逻辑和错误处理代码之间就会有良好的区隔。大量代码会开始变得像是整洁而简朴的算法。然而，这样做却把错误检测推到了程序的边缘地带。你打包了外部API以抛出自己的异常，你在代码的顶端定义了一个处理器来应付任何失败了的运算。在大多数时候，这种手段很棒，不过有时你也许不愿这么做。



来看一个例子。下面的笨代码来自某个记账应用的开支总计模块：

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.get
ID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

业务逻辑是，如果消耗了餐食，则计入总额中。如果没有消耗，则员工得到当日餐食补贴。异

常打断了业务逻辑。如果不去处理特殊情况会不会好一些？那样的话代码看起来会更简洁。就像这样：

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

能把代码写得那样简洁吗？能。可以修改ExpenseReportDAO，使其总是返回MealExpense对象。如果没有餐食消耗，就返回一个返回餐食补贴的MealExpense对象。

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // return the per diem default
  }
}
```

这种手法叫做特例模式（SPECIAL CASE PATTERN，[Fowler]）。创建一个类或配置一个对象，用来处理特例。你来处理特例，客户代码就不用应付异常行为了。异常行为被封装到特例对象中。

7.7　别返回null值

我认为，要讨论错误处理，就一定要提及那些容易引发错误的做法。第一项就是返回null值。我不想去计算曾经见过多少几乎每行代码都在检查null值的应用程序。下面就是个例子：

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

这种代码看似不坏，其实糟透了！返回null值，基本上是在给自己增加工作量，也是在给调用者添乱。只要有一处没检查null值，应用程序就会失控。

你有没有注意到，嵌套if语句的第二行没有检查null值？如果在运行时persistentStore为null会发生什么事？我们会在运行时得到一个NullPointerException异常，也许有人在代码顶端捕

获这个异常，也可能没有捕获。两种情况都很糟糕。对于从应用程序深处抛出的NullPointerException异常，你到底该作何反应呢？

可以敷衍说上列代码的问题是少做了一次null值检查，其实问题多多 。如果你打算在方法中返回null值，不如抛出异常，或是返回特例对象。如果你在调用某个第三方API中可能返回null值的方法，可以考虑用新方法打包这个方法，在新方法中抛出异常或返回特例对象。

在许多情况下，特例对象都是爽口良药。设想有这么一段代码：

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

现在，getExployees可能返回null，但是否一定要这么做呢？如果修改getEmployee，返回空列表，就能使代码整洁起来：

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

所幸Java有Collections.emptyList( )方法，该方法返回一个预定义不可变列表，可用于这种目的：

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

这样编码，就能尽量避免NullPointerException的出现，代码也就更整洁了。

7.8 别传递null值

在方法中返回null值是糟糕的做法，但将null值传递给其他方法就更糟糕了。除非API要求你向它传递null值，否则就要尽可能避免传递null值。

举例说明原因。用下面这个简单的方法计算两点的投射：

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  …
}
```

如果有人传入null值会怎样？

```
calculator.xProjection(null, new Point(12, 13));
```

当然，我们会得到一个NullPointerException异常。

如何修正？可以创建一个新异常类型并抛出：

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x – p1.x) * 1.5;
  }
}
```

这样做好些吗？可能比null指针异常好一些，但要记住，我们还得为InvalidArgumentException异常定义处理器。这个处理器该做什么？还有更好的做法吗？

还有替代方案。可以使用一组断言：

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x – p1.x) * 1.5;
  }
}
```

看上去很美，但仍未解决问题。如果有人传入null值，还是会得到运行时错误。

在大多数编程语言中，没有良好的方法能对付

由调用者意外传入的null值。事已如此，恰当的做法就是禁止传入null值。这样，你在编码的时候，就会时时记住参数列表中的null值意味着出问题了，从而大量避免这种无心之失。

7.9　小结

整洁代码是可读的，但也要强固。可读与强固并不冲突。如果将错误处理隔离看待，独立于主要逻辑之外，就能写出强固而整洁的代码。做到这一步，我们就能单独处理它，也极大地提升了代码的可维护性。

7.10 文献

[Martin]：*Agile Software Development: Principles, Patterns, and Practices* , Robert C. Martin, Prentice Hall, 2002.

---

[1] 原注：[Martin]。

第8章　边界

James Grenning



我们很少控制系统中的全部软件。有时我们购买第三方程序包或使用开放源代码，有时我们依靠公司中其他团队打造组件或子系统。不管是哪种情况，我们都得将外来代码干净利落地整合进自己的代码中。本章将介绍一些保持软件边界整洁的实践手段和技巧。

8.1　使用第三方代码

在接口提供者和使用者之间，存在与生俱来的张力。第三方程序包和框架提供者追求普适性，这样就能在多个环境中工作，吸引广泛的用户。而使用者则想要集中满足特定需求的接口。这种张力会导致系统边界上出现问题。

以java.util.Map为例。如你在表8-1中所见，Map有着广阔的接口和丰富的功能。当然，这种力量和灵活性很有用，但也要付出代价。比如，应用程序可能构造一个Map对象并传递它。我们的初衷可能是Map对象的所有接收者都不要删除映射图中的任何东西。但表8-1的顶端却正好有一个clear( )方法。Map的任何使用者都能清除映射图。或许设计惯例是Map中只能保存特定的类型，但Map并不会可靠地约束存于其中的对象的类型。使用者可随意往Map中塞入任何类型的条目。

```
clear() void - Map
containsKey(Object key) boolean - Map
containsValue(Object value) boolean - Map
entrySet() Set - Map
equals(Object o) boolean - Map
get(Object key) Object - Map
getClass() Class<? extends Object> - Object
hashCode() int - Map
isEmpty() boolean - Map
keySet() Set - Map
notify() void - Object
notifyAll() void - Object
put(Object key, Object value) Object - Map
putAll(Map t) void - Map
remove(Object key) Object - Map
size() int - Map
toString() String - Object
values() Collection - Map
wait() void - Object
wait(long timeout) void - Object
wait(long timeout, int nanos) void - Object
```

图8-1　Mop类的方法

如果你的应用程序需要一个包容Sensor类对象的Map映射图，大概会是这样：

```
Map sensors = new HashMap();
```

当代码的其他部分需要访问这些sensor，就会

有这行代码：

```
Sensor s = (Sensor)sensors.get(sensorId  );
```

这行代码一再出现。代码的调用端承担了从Map中取得对象并将其转换为正确类型的职责。行倒是行，却并非整洁的代码。而且，这行代码并未说明自己的用途。通过对泛型的使用，这段代码可读性可以大大提高，如下所示：

```
  Map<Sensor> sensors = new HashMap<Sensor>();
...
  Sensor s = sensors.get(sensorId );
```

不过，Map<Sensor>提供了超出所需/所愿的功能的问题，仍未得到解决。

在系统中不受限制地传递Map<Sensor>的实体，意味着当到Map的接口被修改时，有许多地方都要跟着改。你或许会认为这样的改动不太可能发生，不过，当Java 5加入对泛型的支持时，的确发生了改动。我们也的确见到一些系统因为要做大量改动才能自由使用Map类，而无法使用泛型。

使用Map的更整洁的方式大致如下。Sensors的

用户不必关心是否用了泛型，那将是（也该是）实现细节才关心的。

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }

  //片段
}
```

边界上的接口（Map）是隐藏的。它能随来自应用程序其他部分的极小的影响而变动。对泛型的使用不再是个大问题，因为转换和类型管理是在Sensors类内部处理的。

该接口也经过仔细修整和归置以适应应用程序的需要。结果就是得到易于理解、难以被误用的代码。Sensors类推动了设计和业务的规则。

我们并不建议总是以这种方式封装Map的使用。我们建议不要将Map（或在边界上的其他接口）在系统中传递。如果你使用类似Map这样的边界接口，就把它保留在类或近亲类中。避免从公共API中返回边界接口，或将边界接口作为参数传递给公共API。

8.2　浏览和学习边界

第三方代码帮助我们在更少时间内发布更丰富的功能。在利用第三方程序包时，该从何处入手呢？我们没有测试第三方代码的职责，但为要使用的第三方代码编写测试，可能最符合我们的利益。

设想第三方代码库的使用方法并不清楚。我们可能会花上一两天（或者更多）时间阅读文档，决定如何使用。然后，我们会编写使用第三方代码的代码，看看是否如我们所愿地工作。陷入长时间的调试、找出在我们或他们代码中的缺陷，这可不是什么稀罕事。

学习第三方代码很难。整合第三方代码也很难。同时做这两件事难上加难。如果我们采用不同的做法呢？不要在生产代码中试验新东西，而是编写测试来遍览和理解第三方代码。Jim Newkirk把这叫做学习性测试（learning tests）[1]。

在学习性测试中，我们如在应用中那样调用第三方代码。我们基本上是在通过核对试验来检测自己对那个API的理解程度。测试聚焦于我们想从API得到的东西。

8.3 学习log4j

比如，我们想使用apache log4j包来代替自定义的日志代码。我们下载了log4j，打开介绍文档页。无需看太久，就编写了第一个测试用例，希望它能向控制台输出hello字样。

```
@Test
public void testLogCreate() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.info("hello");
}
```

运行，logger发生了一个错误，告诉我们需要用Appender。再多读一点文档，我们发现有个ConsoleAppender。于是我们创建了一个ConsoleAppender，再看是否能解开向控制台输出日志的秘诀。

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  ConsoleAppender appender = new ConsoleAppender();
  logger.addAppender(appender);
  logger.info("hello");
}
```

这回，我们发现Appender没有输出流。奇怪，它该有输出流的。在Google上得到一点帮助后，我们写了以下代码：

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.removeAllAppenders();
  logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n"),
      ConsoleAppender.SYSTEM_OUT));
  logger.info("hello");
}
```

这回行了；hello字样的日志信息出现在控制台上！必须告知ConsoleAppender，让它往控制台写字，看起来有点奇怪。

很有趣，当我们移除ConsoleAppender.SystemOut参数时，那个hello字样仍然输出到屏幕上。但如果取走PatternLayout，就会出现关于没有输出流的错误信息。这实在太古怪了。

再仔细看看文档，我们看到默认的ConsoleAppender构造器是“未配置”的，这看起来并不明显或没什么用，反而像是log4j的一个缺陷，或者至少是前后不太一致。

再搜索、阅读、测试，最终我们得到代码清单8-1。我们极大地发掘了log4j的工作方式，也将得到的知识融入了一系列简单的单元测试中。

代码清单8-1　LogTest.java

```
public class LogTest {
  private Logger logger;

  @Before
  public void initialize() {
      logger = Logger.getLogger("logger");
      logger.removeAllAppenders();
      Logger.getRootLogger().removeAllAppenders();
  }
  @Test
  public void basicLogger() {
      BasicConfigurator.configure();
      logger.info("basicLogger");
  }

  @Test
  public void addAppenderWithStream() {
   logger.addAppender(new ConsoleAppender(
       new PatternLayout("%p %t %m%n"),
       ConsoleAppender.SYSTEM_OUT));
     logger.info("addAppenderWithStream");
  }

  @Test
  public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
       new PatternLayout("%p %t %m%n")));
     logger.info("addAppenderWithoutStream");
  }
}
```

现在我们知道如何初始化一个简单的控制台日

志器，也能把这些知识封装到自己的日志类中，好将应用程序的其他部分与log4j的边界接口隔离开来。

8.4　学习性测试的好处不只是免费

学习性测试毫无成本。无论如何我们都得学习要使用的API，而编写测试则是获得这些知识的容易而不会影响其他工作的途径。学习性测试是一种精确试验，帮助我们增进对API的理解。

学习性测试不光免费，还在投资上有正面的回报。当第三方程序包发布了新版本，我们可以运行学习性测试，看看程序包的行为有没有改变。

学习性测试确保第三方程序包按照我们想要的方式工作。一旦整合进来，就不能保证第三方代码总与我们的需要兼容。原作者不得不修改代码来满足他们自己的新需要。他们会修正缺陷、添加新功能。风险伴随新版本而来。如果第三方程序包的修改与测试不兼容，我们也能马上发现。

无论你是否需要通过学习性测试来学习，总要有一系列与生产代码中调用方式一致的输出测试来支持整洁的边界。不使用这些边界测试来减轻迁移的劳力，我们可能会超出应有时限，长久地绑在旧版本上面。

8.5　使用尚不存在的代码

还有另一种边界，那种将已知和未知分隔开的边界。在代码中总有许多地方是我们的知识未及之处。有时，边界那边就是未知的（至少目前未知）。有时，我们并不往边界那边看过去。

好多年以前，我曾在一个开发无线通信系统软件的团队中工作。该系统有个子系统Transmitter（发送机）。我们对Transmitter知之甚少，而该子系统的开发者还没有对接口进行定义。我们不想受这种事阻碍，就从距未知那部分代码很远处开始工作。

对于我们的世界如何结束、新世界如何开始，我们有许多好主意。工作时，我们偶尔会跨越那道边界。尽管云雾遮挡了我们看向边界那边的视线，我们还是从工作中了解到我们想要 的边界接口是什么样的。我们想要告知发送机一些事：

将发送机置于指定频率，并发出自这个流得到的数据的模拟表示。

我们不知这会如何做到，因为API还没设计出来。所以，我们决定过后再编写细节代码。

为了不受阻碍，我们定义了自己使用的接口。我们给它取了个好记的名字，比如Transmitter。我们给它写了个名为transmit的方法，获取频率参数和数据流。这就是我们希望得到 的接口。

编写我们想得到的接口，好处之一是它在我们控制之下。这有助于保持客户代码更可读，且集中于它该完成的工作。

在图8-2中可以看到，我们将CommunicationsController类从发送器API（该API不受我们控制，而且还没定义）中隔离出来。通过使用符合应用程序的接口，CommunicationsController代码整洁且足以表达其意图。一旦发送器API被定义出来，我们就编写TransmitterAdapter来跨接。ADAPTER [2] 封装了与API的互动，也提供了一个当API发生变动时唯一需要改动的地方。

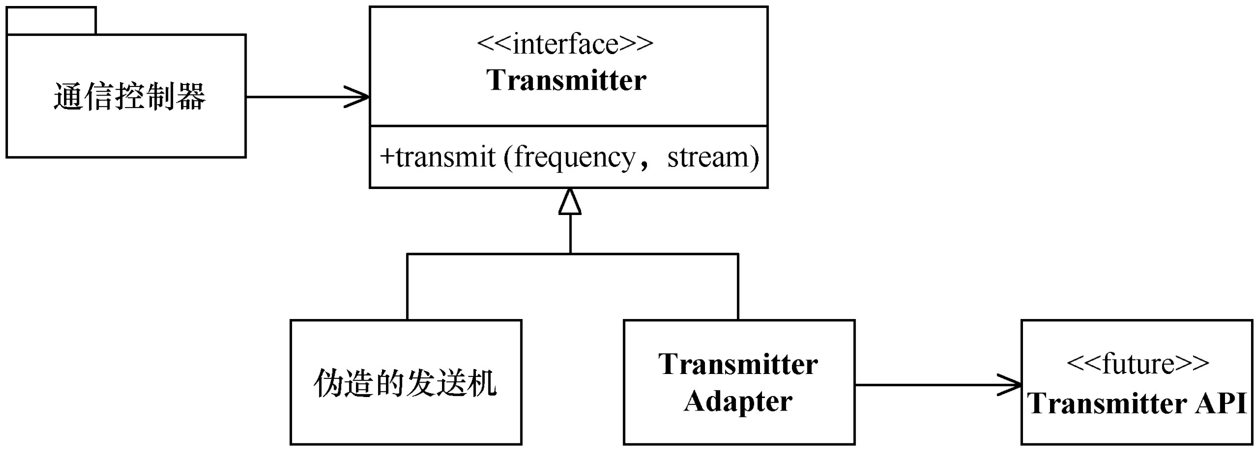


图8-2　对发送器的预测

这套设计方案为测试提供了一种极为方便的接缝[3]。使用适当的FakeTransmitter，我们就能测试CommunicationsController类。在拿到TransmitterAPI时，我们也能创建确保正确使用API的边界测试。

8.6　整洁的边界

边界上会发生有趣的事。改动是其中之一。有良好的软件设计，无需巨大投入和重写即可进行修改。在使用我们控制不了的代码时，必须加倍小心保护投资，确保未来的修改不至于代价太大。

边界上的代码需要清晰的分割和定义了期望的测试。应该避免我们的代码过多地了解第三方代码中的特定信息。依靠你 能控制的东西，好过依靠你控制不了的东西，免得日后受它控制。

我们通过代码中少数几处引用第三方边界接口的位置来管理第三方边界。可以像我们对待Map那样包装它们，也可以使用ADAPTER模式将我们的接口转换为第三方提供的接口。采用这两种方式，代码都能更好地与我们沟通，在边界两边推动内部一致的用法，当第三方代码有改动时修改点也会更少。

8.7 文献

[BeckTDD]：*Test Driven Development*, Kent Beck, Addison-Wesley, 2003.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[WELC]：*Working Effectively with Legacy Code*, Addison-Wesley, 2004.

---

[1] 原注：[BeckTDD], pp. 136–137。

[2] 原注：见[GOF]中的Adapter模式。

[3] 原注：在[WELC]中可查阅更多关于接缝（seam）的信息。

第9章　单元测试



过去十年以来，编程专业领域进步很大。1997年时，没人听说过测试驱动开发。对于我们之中的大多数人来说，单元测试是那种用来确保程序“可运行”的用过即扔的短代码。我们辛勤地编写类和方法，再弄出一些特殊代码来测试它们。通常这会是种简单的驱动式程序，让我们能够手工与自己编写的程序交互。

我记得在20世纪90年代曾为一套嵌入式实时系

统编写过C++程序。该程序是个简单的计时器，有如下签名：

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

想法很简单；到达指定毫秒数时，在一个新线程中执行Command的excute方法。问题在于如何测试它。

我随便写了个简单的驱动式程序，聆听来自键盘的动作。键盘输入一个字符时，它就安排5秒钟之后输出同样的字符。我输入了一句带节奏的歌词，然后等着5秒钟之后它在屏幕上重现出来。

I . . . want-a-girl . . . just . . . like-the-girl-who-marr . . . ied . . . dear . . . old . . . dad. [1]

在按下那些“.”键时，我真的在哼着那段旋律，当那些句点出现在屏幕上时，我又再哼了一次。

那就是我的测试！我看到这法子可行，演示给同事们看，然后就把代码扔掉了。

如前文所述，我们的专业领域进步甚多。如今，我会编写测试，确保代码中每个犄角旮旯都如

我所愿地工作。我会将代码和操作系统隔离开，而不是直接调用标准计时功能。我会伪造一套计时函数，这样就能全面控制时间。我会安排一些设置布尔值标识的命令，往前步进时间，查看这些标识，确保它们在我将时间调到正确值时由false变为true。

有了一套运行通过的测试，我会确保任何需要用到代码的人都能方便地使用这些测试。我会确保测试和代码一起签入同一个代码包。

对，我们进步甚多；但还有很长的路要走。敏捷和TDD运动鼓舞了许多程序员编写自动化单元测试，每天还有更多人加入这个行列。但是，在争先恐后将测试加入规程中时，许多程序员遗漏了一些关于编写好测试的更细微但却重要的要点。

9.1　TDD三定律

谁都知道TDD要求我们在编写生产代码前先编写单元测试。但这条规则只是冰山之巅。看看下列三定律[2]：

定律一　在编写不能通过的单元测试前，不可编写生产代码。

定律二　只可编写刚好无法通过的单元测试，不能编译也算不通过。

定律三　只可编写刚好足以通过当前失败测试的生产代码。

这三条定律将你限制在大概30秒一个的循环中。测试与生产代码一起 编写，测试只比生产代码早写几秒钟。

这样写程序，我们每天就会编写数十个测试，每个月编写数百个测试，每年编写数千个测试。这样写程序，测试将覆盖所有生产代码。测试代码量足以匹敌生产代码量，导致令人生畏的管理问题。

9.2 保持测试整洁

几年前，有人请我去指导一个开发团队。那个团队认定，测试代码的维护不应 遵循生产代码的质量标准。他们彼此默许在单元测试中破坏规矩。“速而不周”成了团队格言。变量命名不用好，测试函数不必短小和具有描述性。测试代码不必做良好设计和仔细划分。只要测试代码还能工作，只要还覆盖着生产代码，就足够好。

有些读者可能会同意这种做法。或许，在很久以前，你也用过我为那个Timer类写测试的方法。从编写那种用后即扔的测试到编写全套自动化单元测试是一大进步。所以，就像那个我指导过的团队一样，你或许也会认为脏测试好过没测试。

这个团队没有意识到的是，脏测试等同于——如果不是坏于的话——没测试。问题在于，测试必须随生产代码的演进而修改。测试越脏，就越难修改。测试代码越缠结，你就越有可能花更多时间塞进新测试，而不是编写新生产代码。修改生产代码后，旧测试就会开始失败，而测试代码中乱七八糟的东西将阻碍代码再次通过。于是，测试变得就像是不断翻番的债务。

随着版本递进，团队维护测试代码组的代价也在上升。最终，它变成了开发者最大的抱怨对象。当经理们问及为何超支如此巨大，开发者们就归咎于测试。最后，他们只能扔掉了整个测试代码组。

但是，没有了测试代码组，他们就失去了确保对代码的改动能如愿工作的能力。没有了测试代码组，他们就无法确保对系统某个部分的修改不会影响到系统的其他部分。故障率开始增加。随着并非出自有意的故障越来越多，他们开始害怕做改动。他们不再清理生产代码，因为他们害怕修改带来的损害多于收益。生产代码开始腐坏。最后，他们只剩下没有测试、纷乱而缺陷缠身的生产代码，沮丧的客户，还有对测试的失望。

在某种意义上，他们说对了。测试的确 让他们失望。不过是他们自己决定让测试变得乱七八糟的，而那正是失败的根源。如果他们保持测试整洁，测试就不会令他们失望。我可以拍着胸脯这么说，因为我曾经参与并指导了多个凭借整洁 单元测试获得成功的团队。

故事的寓意很简单：测试代码和生产代码一样重要 。它可不是二等公民。它需要被思考、被设计和被照料。它该像生产代码一般保持整洁。

测试带来一切好处

如果测试不能保持整洁，你就会失去它们。没有了测试，你就会失去保证生产代码可扩展的一切要素。你没看错。正是单元测试 让你的代码可扩展、可维护、可复用。原因很简单。有了测试，你就不担心对代码的修改！没有测试，每次修改都可能带来缺陷。无论架构多有扩展性，无论设计划分得有多好，没有了测试，你就很难做改动，因为你担忧改动会引入不可预知的缺陷。

有了测试，愁云一扫而空。测试覆盖率越高，你就越不担心。哪怕是对于那种架构并不优秀、设计晦涩纠缠的代码，你也能近乎没有后患地做修改。实际上，你能毫无顾虑地改进 架构和设计！

所以，覆盖了生产代码的自动化单元测试程序组能尽可能地保持设计和架构的整洁。测试带来了一切好处，因为测试使改动变得可能。

如果测试不干净，你改动自己代码的能力就有所牵制，而你也会开始失去改进代码结构的能力。测试越脏，代码就会变得越脏。最终，你丢失了测试，代码开始腐坏。

9.3　整洁的测试

整洁的测试有什么要素？有三个要素：可读性，可读性和可读性。在单元测试中，可读性甚至比在生产代码中还重要。测试如何才能做到可读？和其他代码中一样：明确，简洁，还有足够的表达力。在测试中，你要以尽可能少的文字表达大量内容。

来看看代码清单9-1中来自FitNesse的代码。这三个测试很难读懂，显然有改善空间。首先，其中有数量恐怖的重复代码[G5]调用addPage和assertSubString。更重要的是，代码中充满了干扰测试表达力的细节。

代码清单9-1　SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
  String xml = response.getContent();
```

```
  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks
()
throws Exception
{
  WikiPage pageOne = crawler.addPage(root, PathParser.parse("Pag
eOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
  WikiPageProperty symLinks = properties.set(SymbolicPage.PROPER
TY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
  crawler.addPage(root, PathParser.parse("TestPageOne"), "test p
age");

  request.setResource("TestPageOne");
  request.addInput("type", "data");
  Responder responder = new SerializedPageResponder();
```

```
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("test page", xml);
  assertSubString("<Test", xml);
}
```

请看对PathParser的那些调用。它们将字符串转换为供爬虫使用的PagePath实体。转换与测试毫无关系，徒然混淆了代码的意图。与创建responder相关的细节，还有response的收集与转换也尽是噪声。此外还有从resource和参数构造请求URL的笨手段。（这些代码我有幸参与编写，所以可以敞开来批评。）

最终，这段代码不是设计来给人看的。可怜的读者淹没在细节的汪洋大海中，在真正用到测试之前，还得理解这些细节。

现在看看代码清单9-2中改进了的测试。这些测试还是做一样的事，不过已经被重构为更整洁和有表达力的形式。

代码清单9-2　SerializedPageResponderTest.java（重构后）

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");
```

```
  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>Chi
ldOne</name>"
  );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws
Exception {
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");

  addLinkTo(page, "PageTwo", "SymPage");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>Chi
ldOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");

  submitRequest("TestPageOne", "type:data");

  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

这些测试显然呈现了构造-操作-检验（BUILD-OPERATE-CHECK）[3] 模式。每个测试都清晰地拆分为三个环节。第一个环节构造测试数据，第二

个环节操作测试数据，第三个部分检验操作是否得到期望的结果。

注意，那些恼人的细节大部分消失了。测试直达目的，只用到那些真正需要的数据类型和函数。读测试的人应该都能够很快搞清楚状况，不至于被细节误导或吓倒。

9.3.1 面向特定领域的测试语言

代码清单9-2中的测试展示了为测试构造一种面向特定领域的语言的技巧。我们没有直接使用程序员用来对系统进行操作的API，而是打造了一套包装这些API的函数和工具代码，这样就能更方便地编写测试，写出来的测试也更便于阅读。那正是一种测试语言 ，可以帮助程序员编写自己的测试，也可以帮助后来者阅读测试。

这种测试API并非起初就设计出来，而是在对那些充满令人迷惑细节的测试代码进行后续重构时逐渐演进。如同你看见我将代码清单9-1重构为代码清单9-2一般，守规矩的开发者也将他们的测试代码重构为更简洁和具有表达力的形式。

9.3.2 双重标准

在某种意义上，本章开始处提到的那个团队的做法是正确的。测试API中的代码与生产代码相比，的确 有一套不同的工程标准。测试代码应当简单、精悍、足具表达力，但它该和生产代码一般有效。毕竟它是在测试环境而非生产环境中运行，这两种环境有着截然不同的需求。

请看代码清单9-3中的测试。在为某个环境控制系统设计原型时，我写了这个测试。无需深入细节，你就能说出该测试在“温度太低”时检验温度警报器、加热器和送风机是否全部打开。

代码清单9-3　EnvironmentControllerTest.java

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

当然，这里头也有许多细节。例如，tic函数是做什么的？实际上，在读测试时你可以不用担心这些问题。你只需考虑是否同意系统最终状态是否与“温度太低”的情况相符。

当你阅读这个测试时，可以留意到自己的眼光得在被检验的状态的名称与状态的“意义 ”之间来回跳转。你看到heaterState，眼光向左滑到assertTrue。你看到coolerState，眼光向左看assertFalse。这个过程既乏味又不可靠。它让测试变得难以阅读。

我大幅改进了测试的可读性，得到代码清单9-4。

代码清单9-4　EnvironmentControllerTest.java（重构后）

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

当然，我创建了一个wayTooCold函数，隐藏了tic函数的细节。不过要注意的是assertEquals中的那个奇怪的字符串。大写表示“打开”，小写表示“关闭”，那些字符遵循以下次序：{heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}。

尽管这破坏了思维映射 [4] 的规则，看来它在这种情况下还是适用的。只要你明白其含义，你就能一眼看到那个字符串，迅速译解出结果。

代码清单9-5　EnvironmentControllerTest.java（扩展到更大范围）

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception
{
    tooCold();
    assertEquals("HBchl", hw.getState());
  }

  @Test
  public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
  }

  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

代码清单9-6中给出了getState函数。注意，代码效率不是非常高。要提升效率，可能应该使用StringBuffer。

代码清单9-6　MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
```

```
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
  return state;
}
```

StringBuffer有点丑陋。即便在生产代码中，假使代价较小，我都会避免使用StringBuffer；而且你可以看到，清单9-6中代码的代价的确很小。这套应用显然是嵌入式实时系统，计算机和内存资源都很有限。不过，测试 环境大概完全不必做限制。

这就是双重标准。有些事你大概永远不会在生产环境中做，而在测试环境中做却完全没问题。通常这关乎内存或CPU效率的问题，不过却永远不会与整洁有关。

9.4　每个测试一个断言

有个流派 [5] 认为，JUnit中每个测试函数都应该有且只有一个断言语句。这条规则看似过于苛求，但其好处却可以在代码清单9-5中看到。这些测试都归结为一个可快速方便地理解的结论。

代码清单9-2又如何？我们能将关于输出是XML的断言与输出包含某些子字符串的断言轻易地组合到一起，不过这样做看来毫无道理。然而，我们可以将测试分解为两个单独的测试，每个都有自己的断言，如代码清单9-7所示。

代码清单9-7　SerializedPageResponderTest.java（单个断言的版本）

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception
 {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

  whenRequestIsIssued("root", "type:pages");

  thenResponseShouldContain(
     "<name>PageOne</name>", "<name>PageTwo</name>", "<name>Chi
```

```
ldOne</name>"
  );
}
```

注意，我修改了那些函数的名称，以符合given-when-then [6] 约定。这让测试更易阅读。不幸的是，如此分解测试，导致了许多重复代码的出现。

可以利用模板方法（TEMPLATE METHOD）[7] 模式，将given/when部分放到基类中，将then部分放到派生类中，消除代码重复问题。或者，我们也可以创建一个完整的单独测试类，把given和when部分放到@Before函数中，把when部分放到每个@Test函数中。但对于这个小问题，这看来有点太机械。最后，我还是保留了代码清单9-2那种多个断言的形式。

我认为，单个断言是个好准则 [8] 。我通常都会创建支持这条准则的特定领域测试语言，如代码清单9-5所示。不过，我也不害怕在单个测试中放入一个以上断言。我认为，最好的说法是单个测试中的断言数量应该最小化。

每个测试一个概念

更好一些的规则或许是每个测试函数中只测试一个概念。我们不想要超长的测试函数，测试完这个又测试那个。代码清单9-8就是那样一种测试的例子。这个测试应当拆解为3个单独测试，因为它测试了3件不同的事。把三者混到一起，读者就不得不猜想每段代码出现的理由，以及那段代码到底要测试什么。

代码清单9-8

```
/**
 * Miscellaneous tests for the addMonths() method.
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonth
s(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

这三个测试函数大概应该像这个样子：

- 对于某个有31天的月份的最后一天（如五月）：

（1）增加一个该月份最末一天为30日（如六月）的月份时，日期应该是该月的30日而非31日。

（2）增加最末月有31天的两个月时，日期应该是31日。

- 对于某个有30天的月份的最后一天（如六月）：

（3）增加一个有31天的月份时，日期应该是30日而非31日。

这样一来，你可以看到，在这些混杂的测试当中，隐藏有一条普遍规则。增加月份数时，日期不能大于该月份的最末一天。这意味着在2月28日增加月份数，就会得到3月28日。而这个测试应该有用，但被遗漏了。

并非是代码清单9-8中每个段落的多重断言导致问题。问题在于，有多个概念被测试，所以，最佳规则也许是应该尽可能减少每个概念的断言数量，每个测试函数只测试一个概念。

9.5　F.I.R.S.T. [9]

整洁的测试还遵循以下5条规则，这5条规则的首字母构成了本节标题：

快速（Fast）　测试应该够快。测试应该能快速运行。测试运行缓慢，你就不会想要频繁地运行它。如果你不频繁运行测试，就不能尽早发现问题，也无法轻易修正，从而也不能轻而易举地清理代码。最终，代码就会腐坏。

独立（Independent）　测试应该相互独立。某个测试不应为下一个测试设定条件。你应该可以单独运行每个测试，及以任何顺序运行测试。当测试互相依赖时，头一个没通过就会导致一连串的测试失败，使问题诊断变得困难，隐藏了下级错误。

可重复（Repeatable）　测试应当可在任何环境中重复通过。你应该能够在生产环境、质检环境中运行测试，也能够在无网络的列车上用笔记本电脑运行测试。如果测试不能在任意环境中重复，你就总会有个解释其失败的接口。当环境条件不具备时，你也会无法运行测试。

自足验证（Self-Validating） 测试应该有布尔值输出。无论是通过或失败，你不应该查看日志文件来确认测试是否通过。你不应该手工对比两个不同文本文件来确认测试是否通过。如果测试不能自足验证，对失败的判断就会变得依赖主观，而运行测试也需要更长的手工操作时间。

及时（Timely） 测试应及时编写。单元测试应该恰好在使其通过的生产代码之前编写。如果在编写生产代码之后编写测试，你会发现生产代码难以测试。你可能会认为某些生产代码本身难以测试。你可能不会去设计可测试的代码。

9.6　小结

我们只是触及了这个话题的表面。实际上，我认为应该为整洁的测试写上一整本书。对于项目的健康度，测试盒生产代码同等重要。或许测试更为重要，因为它保证和增强了生产代码的可扩展性、可维护性和可复用性。所以，保持测试整洁吧。让测试具有表达力并短小精悍。发明作为面向特定领域语言的测试API，帮助自己编写测试。

如果你坐视测试腐坏，那么代码也会跟着腐坏。保持测试整洁吧。

9.7　文献

[RSpec]：*RSpec: Behavior Driven Development for Ruby Programmers* , Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software* , Gamma et al., Addison-Wesley, 1996.

---

[1]　译注：I want a girl just like the girl who married dear old dad是20世纪初American Quartet四重唱乐队的歌曲名，也是歌词中的一句，这里不做翻译。

[2]　原注：Professionalism and Test-Driven Development, Robert C. Martin, Object Mentor, IEEE Software, May/June 2007 (Vol. 24, No. 3) pp. 32–36。

[3]　原注：http://fitnesse.org/FitNesse.AcceptanceTestPattern。

[4]　原注：见第2章。

[5]　原注：见Dave Astel的blog文章：http://www.artima.com/weblogs/viewpost.jsp?thread=35578 。

[6]　原注：[RSpec]。

[7]　原注：[GOF]。

[8]　原注："照规矩办（Keep to the code）!"【译者按】这是电影《加勒比海盗》中的一句台词。

[9]　原注：参见Object Mentor培训材料。

第10章　类

与Jeff Langr合写



本书到目前为止一直在讨论如何编写良好的代码行和代码块。我们深入研究了函数的恰当构成，以及函数之间如何互相关联。不过，尽管讨论了这么多关于代码语句及由代码语句构成的函数的表达力，除非我们将注意力放到代码组织的更高层面，就始终不能得到整洁的代码。

10.1　类的组织

遵循标准的Java约定，类应该从一组变量列表开始。如果有公共静态常量，应该先出现。然后是私有静态变量，以及私有实体变量。很少会有公共变量。

公共函数应跟在变量列表之后。我们喜欢把由某个公共函数调用的私有工具函数紧随在该公共函数后面。这符合了自顶向下原则，让程序读起来就像一篇报纸文章。

封装

我们喜欢保持变量和工具函数的私有性，但并不执着于此。有时，我们也需要用到受护（protected）变量或工具函数，好让测试可以访问到。对我们来说，测试说了算。若同一程序包内的某个测试需要调用一个函数或变量，我们就会将该函数或变量置为受护或在整个程序包内可访问。然而，我们首先会想办法使之保有隐私。放松封装总是下策。

10.2　类应该短小

关于类的第一条规则是类应该短小。第二条规则是还要更短小。不，我们并不是要重弹“函数”一章的论调。就像函数一样，在设计类时，首要规条就是要更短小。和函数一样，马上有个问题出现，那就是“多小合适呢？”

对于函数，我们通过计算代码行数衡量大小。对于类，我们采用不同的衡量方法，计算权责（responsibility）[1]。

代码清单10-1给出了某个类的轮廓。SuperDashboard类曝露大概70个公共方法。大多数开发者都会同意，这实在是太长了。有些开发者或许会将SuperDashboard类指为“神的类”。

代码清单10-1　权责太多

```
public class SuperDashboard extends JFrame implements MetaDataU
ser
   public String getCustomizerLanguagePath()
   public void setSystemConfigPath(String systemConfigPath)
   public String getSystemConfigDocument()
   public void setSystemConfigDocument(String systemConfigDocum
ent)
   public boolean getGuruState()
   public boolean getNoviceState()
   public boolean getOpenSourceState()
```

```
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMouseSelected)
    public boolean isMouseSelected()
    public LanguageManager getLanguageManager()
    public Project getProject()
    public Project getFirstProject()
    public Project getLastProject()
    public String getNewProjectName()
    public void setComponentSizes(Dimension dim)
    public String getCurrentDir()
    public void setCurrentDir(String newDir)
    public void updateStatus(int dotPos, int markPos)
    public Class[] getDataBaseClasses()
    public MetadataFeeder getMetadataFeeder()
    public void addProject(Project project)
    public boolean setCurrentProject(Project project)
    public boolean removeProject(Project project)
    public MetaProjectHeader getProgramMetadata()
    public void resetDashboard()
    public Project loadProject(String fileName, String projectNa
me)
    public void setCanSaveMetadata(boolean canSave)
    public MetaObject getSelectedObject()
    public void deselectObjects()
    public void setProject(Project project)
    public void editorAction(String actionName, ActionEvent even
t)
    public void setMode(int mode)
    public FileManager getFileManager()
    public void setFileManager(FileManager fileManager)
    public ConfigManager getConfigManager()
    public void setConfigManager(ConfigManager configManager)
    public ClassLoader getClassLoader()
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
```

```
   public MetaObject pasting(
      MetaObject target, MetaObject pasted, MetaProject project
)
   public void processMenuItems(MetaObject metaObject)
   public void processMenuSeparators(MetaObject metaObject)
   public void processTabPages(MetaObject metaObject)
   public void processPlacement(MetaObject object)
   public void processCreateLayout(MetaObject object)
   public void updateDisplayLayer(MetaObject object, int layerI
ndex)
   public void propertyEditedRepaint(MetaObject object)
   public void processDeleteObject(MetaObject object)
   public boolean getAttachedToDesigner()
   public void processProjectChangedState(boolean hasProjectCha
ng)
   public void processObjectNameChanged(MetaObject object)
   public void runProject()
   public void setAçowDragging(boolean allowDragging)
   public boolean allowDragging()
   public boolean isCustomizing()
   public void setTitle(String title)
   public IdeMenuBar getIdeMenuBar()
   public void showHelper(MetaObject metaObject, String propert
yName)
   // ... many non-public methods follow ...
}
```

如果SuperDashboard类只包括代码清单10-2中的方法呢？

代码清单10-2　足够短小了吗？

```
public class SuperDashboard extends JFrame implements MetaDataU
ser{
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
```

```
}
```

5个方法不算多，在这里，虽然方法数量较少，可SuperDashboard还是拥有太多权责 。

类的名称应当描述其权责。实际上，命名正是帮助判断类的长度的第一个手段。如果无法为某个类命以精确的名称，这个类大概就太长了。类名越含混，该类越有可能拥有过多权责。例如，如果类名中包括含义模糊的词，如Processor或Manager或Super，这种现象往往说明有不恰当的权责聚集情况存在。

我们也应该能够用大概25个单词简要描述一个类，且不用“若（if）”、“与（and）”、“或（or）”或者“但（but）”等词汇。我们该如何描述SuperDashboard类呢？“SuperDashboard类提供了对最后拥有焦点的组件的访问能力，我们还能通过它跟踪版本号和构建序列号。”“还能”二字正好提示了SuperDashboard类有太多权责。

10.2.1　单一权责原则

单一权责原则（SRP）[2] 认为，类或模块应有且只有一条加以修改的理由 。该原则既给出了权

责的定义，又是关于类的长度的指导方针。类只应有一个权责——只有一条修改的理由。

代码清单10-2中貌似很小的SuperDashboard类有两条加以修改的理由。首先，它跟踪大概会随软件每次发布而更新的版本信息。第二，它管理Java Swing组件（派生自JFrame，顶层GUI窗口的Swing表现形态）。每次修改Swing代码时，无疑都要更新版本号，但反之未必可行：也可能依据系统中其他代码的修改而更新版本信息。

鉴别权责（修改的理由）常常帮助我们在代码中认识到并创建出更好的抽象。可以轻易地将全部三个处理版本信息的SuperDashboard方法拆解到名为Version的类中（如代码清单10-3所示）。Version类是个极有可能在其他应用程序中得到复用的构造！

代码清单10-3　单一权责类

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

SRP是OO设计中最为重要的概念之一，也是较

为容易理解和遵循的概念之一。奇怪的是SRP往往也是最容易被破坏的类设计原则。经常会遇到做太多事的类。为什么呢？

让软件能工作和让软件保持整洁，是两种截然不同的工作。我们中的大多数人脑力有限，只能更多地把精力放在让代码能工作上，而不是放在保持代码有组织和整洁上。这全然正确。分而治之，其在编程行为 中的重要程度等同于在程序中的重要程度。

问题是太多人在程序能工作时就以为万事大吉了。我们没能把思维转向有关代码组织和整洁的部分。我们直接转向下一个问题，而不是回头将臃肿的类切分为只有单一权责的去耦式单元。

与此同时，许多开发者害怕数量巨大的短小单一目的的类会导致难以一目了然抓住全局。他们认为，要搞清楚一件较大工作如何完成，就得在类与类之间找来找去。

然而，有大量短小类的系统并不比有少量庞大类的系统拥有更多移动部件，其数量大致相等。问题是：你是想把工具归置到有许多抽屉、每个抽屉中装有定义和标记良好的组件的工具箱中呢，还是想要少数几个能随便把所有东西扔进去的抽屉？

每个达到一定规模的系统都会包括大量逻辑和复杂性。管理这种复杂性的首要目标就是加以组织，以便开发者知道到哪儿能找到东西，并且在某个特定时间只需要理解直接有关的复杂性。反之，拥有巨大、多目的类的系统，总是让我们在目前并不需要了解的一大堆东西中艰难跋涉。

再强调一下：系统应该由许多短小的类而不是少量巨大的类组成。每个小类封装一个权责，只有一个修改的原因，并与少数其他类一起协同达成期望的系统行为。

10.2.2 内聚

类应该只有少量实体变量。类中的每个方法都应该操作一个或多个这种变量。通常而言，方法操作的变量越多，就越黏聚到类上。如果一个类中的每个变量都被每个方法所使用，则该类具有最大的内聚性。

一般来说，创建这种极大化内聚类是既不可取也不可能的；另一方面，我们希望内聚性保持在较高位置。内聚性高，意味着类中的方法和变量互相依赖、互相结合成一个逻辑整体。

看看代码清单10-4中一个Stack类的实现方式。

这个类非常内聚。在三个方法中，只有size( )方法没有使用所有两个变量。

代码清单10-4　Stack.java（一个内聚类）

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();

  public int size() {
    return topOfStack;
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

保持函数和参数列表短小的策略，有时会导致为一组子集方法所用的实体变量数量增加。出现这种情况时，往往意味着至少有一个类要从大类中挣扎出来。你应当尝试将这些变量和方法分拆到两个或多个类中，让新的类更为内聚。

10.2.3　保持内聚性就会得到许多短小的类

仅仅是将较大的函数切割为小函数，就将导致更多的类出现。想想看一个有许多变量的大函数。你想把该函数中某一小部分拆解成单独的函数。不过，你想要拆出来的代码使用了该函数中声明的4个变量。是否必须将这4个变量都作为参数传递到新函数中去呢？

完全没必要！只要将4个变量提升为类的实体变量，完全无需传递任何 变量就能拆解代码了。应该很容易 将函数拆分为小块。

可惜这也意味着类丧失了内聚性，因为堆积了越来越多只为允许少量函数共享而存在的实体变量。等一下！如果有些函数想要共享某些变量，为什么不让它们拥有自己的类呢？当类丧失了内聚性，就拆分它！

所以，将大函数拆为许多小函数，往往也是将类拆分为多个小类的时机。程序会更加有组织，也会拥有更为透明的结构。

为了说明我的意思，不如从Knuth的名著Literate Programming（中译版《字面编程》）[3]中摘取一个经过时间考验的例子。代码清单10-5展示了Knuth的PrintPrimes程序的Java版本。为示公平，以下程序并非Knuth原版，而是用他的WEB工

具输出的版本。采用它作为例子的目的，是因为它是展示如何将较大的函数分解为多个较小的函数和类的极好入手点。

代码清单10-5　PrintPrimes.java

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];

    J = 1;
    K = 1;
    P[1] = 2;
    ORD = 2;
    SQUARE = 9;

    while (K < M) {
      do {
        J = J + 2;
        if (J == SQUARE) {
          ORD = ORD + 1;
          SQUARE = P[ORD] * P[ORD];
          MULT[ORD - 1] = J;
```

```
      }
      N = 2;
      JPRIME = true;
      while (N < ORD && JPRIME) {
        while (MULT[N] < J)
          MULT[N] = MULT[N] + P[N] + P[N];
        if (MULT[N] == J)
          JPRIME = false;
        N = N + 1;
      }
    } while (!JPRIME);
    K = K + 1;
    P[K] = J;
  }
  {
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    while (PAGEOFFSET <= M) {
      System.out.println("The First " + M +
                         " Prime Numbers --- Page " + PAGENUM
BER);
      System.out.println("");
      for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR;
 ROWOFFSET++){
        for (C = 0; C < CC;C++)
          if (ROWOFFSET + C * RR <= M)
            System.out.format("%10d", P[ROWOFFSET + C * RR]);
        System.out.println("");
      }
      System.out.println("\f");
      PAGENUMBER = PAGENUMBER + 1;
      PAGEOFFSET = PAGEOFFSET + RR * CC;
    }
  }
 }
}
```

该程序只有一个大函数，简直一团糟。它拥有很深的缩进结构，冗余的变量和紧密耦合的结构。至少应该将其拆分为数个较小的函数。

从代码清单10-6到代码清单10-8，展示了将代码清单10-5中的代码拆分为较小的类和函数，并为这些类、函数和变量取个好名字后的结果。

代码清单10-6　PrimePrinter.java（重构后）

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                               "The First " + NUMBER_OF_PRIMES +

                                  " Prime Numbers");
    tablePrinter.print(primes);
  }
}
```

代码清单10-7　RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;
```

```
public RowColumnPagePrinter(int rowsPerPage,
                            int columnsPerPage,
                            String pageHeader) {
  this.rowsPerPage = rowsPerPage;
  this.columnsPerPage = columnsPerPage;
  this.pageHeader = pageHeader;
  numbersPerPage = rowsPerPage * columnsPerPage;
  printStream = System.out;
}

public void print(int data[]) {
  int pageNumber = 1;
  for (int firstIndexOnPage = 0;
       firstIndexOnPage < data.length;
       firstIndexOnPage += numbersPerPage) {
    int lastIndexOnPage =
      Math.min(firstIndexOnPage + numbersPerPage - 1,
               data.length - 1);
    printPageHeader(pageHeader, pageNumber);
    printPage(firstIndexOnPage, lastIndexOnPage, data);
    printStream.println("\f");
    pageNumber++;
  }
}

private void printPage(int firstIndexOnPage,
                       int lastIndexOnPage,
                       int[] data) {
  int firstIndexOfLastRowOnPage =
    firstIndexOnPage + rowsPerPage - 1;
  for (int firstIndexInRow = firstIndexOnPage;
       firstIndexInRow <= firstIndexOfLastRowOnPage;
       firstIndexInRow++) {
    printRow(firstIndexInRow, lastIndexOnPage, data);
    printStream.println("");
  }
}

private void printRow(int firstIndexInRow,
                      int lastIndexOnPage,
                      int[] data) {
  for (int column = 0; column < columnsPerPage; column++) {
    int index = firstIndexInRow + column * rowsPerPage;
    if (index <= lastIndexOnPage)
      printStream.format("%10d", data[index]);
```

```
    }
  }

  private void printPageHeader(String pageHeader,
                               int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
  }

  public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
  }
}
```

代码清单10-8　PrimeGenerator.java

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirstPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }

  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
```

```
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }

  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate
)) {
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }

  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate)
{
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.s
ize()];
    int leastRelevantMultiple = nextLargerPrimeFactor * nextLarg
erPrimeFactor;
    return candidate == leastRelevantMultiple;
  }

  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
    return true;
  }

  private static boolean
  isMultipleOfNthPrimeFactor(int candidate, int n) {
    return
      candidate == smallestOddNthMultipleNotLessThanCandidate(ca
ndidate, n);
  }

  private static int
  smallestOddNthMultipleNotLessThanCandidate(int candidate, int
n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
```

```
      multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
  }
}
```

你可能注意到的第一件事就是程序比原来长了许多，从1页多增加到了将近3页。这有几个原因。其一，重构后的程序采用了更长、更有描述性的变量名。其二，重构后的程序将函数和类声明当作是给代码添加注释的一种手段。其三，我们采用了空格和格式技巧让程序更可读。

留意程序是如何被拆分为3个主要权责的。PrimePrinter类中只有主程序。主程序的权责是处理执行环境。如果调用方式改变，它也会随之改变。例如，如果程序被转换为SOAP服务，则该类也会被影响到。

RowColumnPagePrinter类懂得如何将数字列表格式化到有着固定行、列数的页面上。若输出格式需要改动，则该类也会被影响到。

PrimeGenerator类懂得如何生成素数列表。注意，这并不意味着要实体化为对象。该类就是个有用的作用域，在其中声明并隐藏变量。如果计算素数的算法发生改动，则该类也会改动。

这并不算是重写！我们没从头开始写一遍程序。实际上，如果你仔细看上述两个不同的程序，就会发现它们采用了同样的算法和机制来完成工作。

我们通过编写验证第一个程序的精确 行为的用例来实现修改。然后，我们做了许多小改动，每次改动一处。每改动一次，就执行一次，确保程序的行为没有变化。一小步接着一小步，第一个程序被逐渐清理和转换为第二个程序。

10.3　为了修改而组织

对于多数系统，修改将一直持续。每处修改都让我们冒着系统其他部分不能如期望般工作的风险。在整洁的系统中，我们对类加以组织，以降低修改的风险。

代码清单10-9中的Sql类用来生成提供恰当元数据的SQL格式化字符串。这个类还没写完，所以暂时不支持update语句等SQL功能。当需要Sql类支持update语句时，我们就得“打开”这个类进行修改。打开类带来的问题是风险随之而来。对类的任何修改都有可能破坏类中的其他代码。必须全面重新测试。

代码清单10-9　一个必须打开修改的类

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] co
lumns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
```

```
}
```

当增加一种新语句类型时，就要修改Sql类。改动单个语句类型时，也要进行修改，比如打算让select功能支持子查询。存在两个修改的理由，说明Sql违反了SRP原则。

可以从一条简单的组织性观点发现对SRP的违反。Sql的方法大纲显示，存在类似selectWithCriteria等只与select语句有关的私有方法。

出现了只与类的一小部分有关的私有方法行为，意味着存在改进空间。然而，展开行动的基本动因却应该是系统的变动。若我们认为Sql类在逻辑上已具足，则无需担心对权责的拆分。如果在可预见的未来无需增加update功能，就该不去动Sql类。不过，一旦打开了类，就应当修正设计方案。

代码清单10-10中的解决方式如何呢？代码清单10-9中Sql类的每个接口方法都重构到从Sql类派生出来的类中了。注意那些私有方法，如valuesList，直接移到了需要用它们的地方。公共私有行为被划分到独立的两个工具类Where和ColumnList中。

代码清单10-10　一组封闭类

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}

public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[] fiel
ds)
  @Override public String generate()
  private String valuesList(Object[] fields, final Column[] colu
mns)
}

public class SelectWithCriteriaSql extends Sql {
  public SelectWithCriteriaSql(
     String table, Column[] columns, Criteria criteria)
  @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
  public SelectWithMatchSql(
     String table, Column[] columns, Column column, String patt
ern)
  @Override public String generate()
}

public class FindByKeySql extends Sql{
  public FindByKeySql(
     String table, Column[] columns, String keyColumn, String k
eyValue)
  @Override public String generate()
}
```

```
public class PreparedInsertSql extends Sql {
  public PreparedInsertSql(String table, Column[] columns)
  @Override public String generate(){
  private String placeholderList(Column[] columns)
}

public class Where {
  public Where(String criteria)
  public String generate()

}
public class ColumnList {
  public ColumnList(Column[] columns)
  public String generate()
}
```

每个类中的代码都变得极为简单。理解每个类花费的时间缩减到近乎为零。函数对其他函数造成毁坏的风险也变得几近于无。从测试的角度看，验证方案中每一处逻辑都成了极为简单的任务，因为类与类之间相互隔离了。

当需要增加update语句时，现存类无需做任何修改，这也同等重要！我们在Sql类的新子类UpdateSql中构建update语句的逻辑。系统中的其他代码都不会因为这个修改而被破坏。

重新架构的Sql逻辑百利而无一弊。它支持SRP。它也支持其他面向对象设计的关键原则，如开放-闭合原则（OCP）[4]：类应当对扩展开放，

对修改封闭。通过子类化手段，重新架构的Sql类对添加新功能是开放的，而且可以同时不触及其他类。只要将UpdateSql类放置到位就行了。

我们希望将系统打造成在添加或修改特性时尽可能少惹麻烦的架子。在理想系统中，我们通过扩展系统而非修改现有代码来添加新特性。

隔离修改

需求会改变，所以代码也会改变。在OO 101 [5] 中，我们学习到，具体类包含实现细节（代码），而抽象类则只呈现概念。依赖于具体细节的客户类，当细节改变时，就会有风险。我们可以借助接口和抽象类来隔离这些细节带来的影响。

对具体细节的依赖给对系统的测试带来了挑战。如果我们构建一个依赖于外部TokyoStockExchange API的Portfolio类，代表投资组合的价值，则测试用例就会受到价值查询的连带影响。如果每5分钟就有新说法，就很难写出测试来。

与其设计直接依赖于TokyoStockExchange的Portfolio类，不如创建StockExchange接口，其中只声明一个方法：

```
public interface StockExchange {
  Money currentPrice(String symbol);
}
```

我们设计TokyoStockExchange类来实现这个接口。我们还要确保Portfolio的构造器接受作为参数的StockExchange引用：

```
public  Portfolio  {
  private  StockExchange  exchange;
  public  Portfolio(StockExchange  exchange)  {
    this.exchange  =  exchange;
  }
  //  ...
}
```

现在就可以为StockExchange接口创建可测试的尝试性实现了。该尝试性实现将返回固定的现值。如果测试中购买了5股微软股票，则尝试性实现总是返回每股100美元的现值。对于StockExchange接口的尝试性实现简化为简单的表格查找。然后再编写一个总投资价值为500美元的测试。

```
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;
  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
```

```
  }
  @Test
  public void GivenFiveMSFTTotalShouldBe500() throws Exception {
    portfolio.add(5, "MSFT");
    Assert.assertEquals(500, portfolio.value());
  }
}
```

如果系统解耦到足以这样测试的程度，也就更加灵活，更加可复用。部件之间的解耦代表着系统中的元素互相隔离得很好。隔离也让对系统每个元素的理解变得更加容易。

通过降低连接度，我们的类就遵循了另一条类设计原则，依赖倒置原则（Dependency Inversion Principle，DIP）[6]。本质而言，DIP认为类应当依赖于抽象而不是依赖于具体细节。

我们的Portfolio类不再依赖于TokyoStockExchange类的实现细节，而是依赖于StockExchange接口。StockExchange接口呈现的是有关询问某只股票价格的抽象概念。这种抽象隔离了所有询价的特定细节，包括价格数据来自何处之类。

10.4 文献

[RDD]： *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

[PPP]： *Agile Software Development* : *Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]： *Literate Programming* , Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.

---

[1] 原注：[RDD]。

[2] 原注：你可在[PPP]中读到更多信息。

[3] 原注：[Knuth92]。

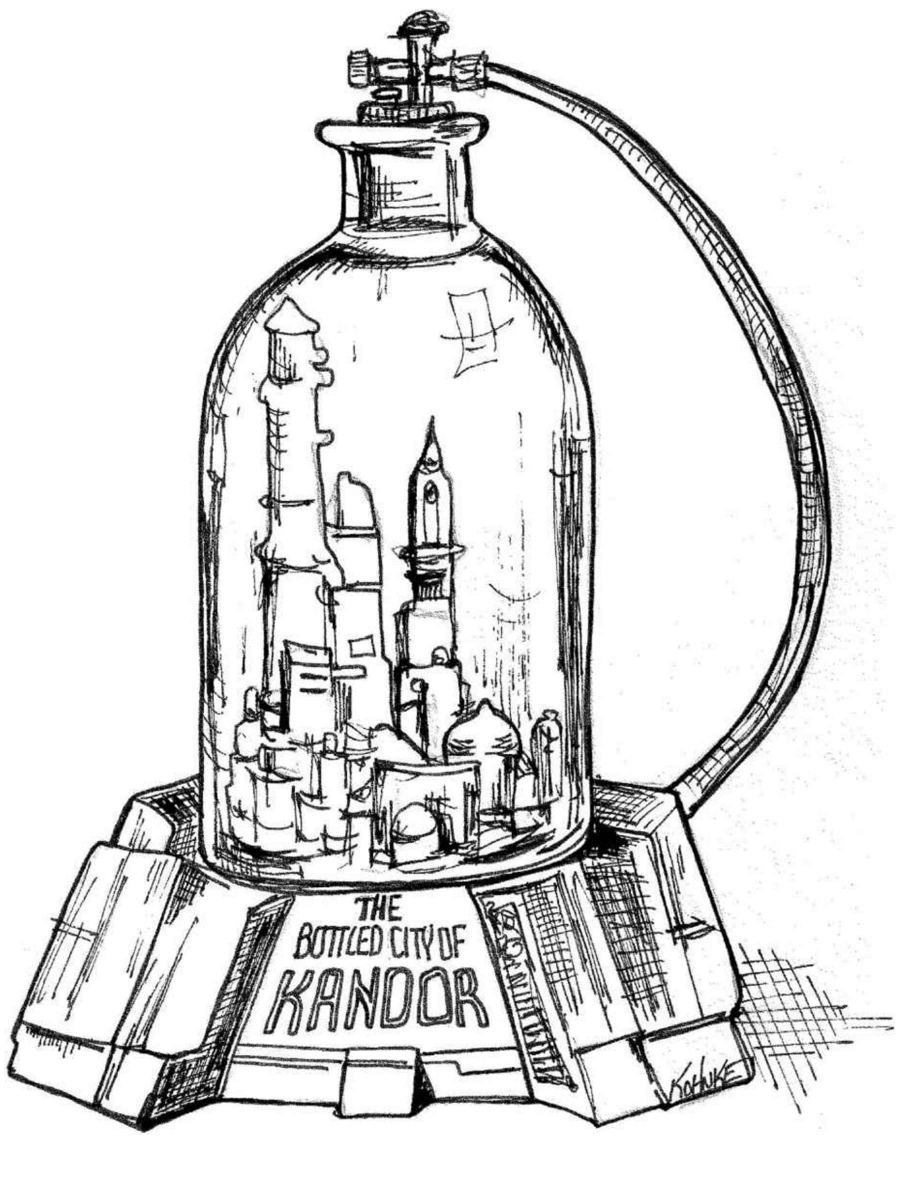
[4] 原注：[PPP]。

[5] 译注：即面向对象入门知识。

[6] 原注：[PPP]。

第11章　系统

Kevin Dean Wampler博士



“复杂要人命。它消磨开发者的生命，让产品难以规划、构建和测试。”

——Ray Ozzie，微软公司首席技术官

11.1　如何建造一个城市

你能自己掌管一切细节吗？大概不行。即便是管理一个既存的城市，也是一个人无法做到的。不过，城市还是在运转（多数时候）。因为每个城市都有一组组人管理不同的部分，供水系统、供电系统、交通、执法、立法，诸如此类。有些人负责全局，其他人负责细节。

城市能运转，还因为它演化出恰当的抽象等级和模块，好让个人和他们所管理的“组件”即便在不了解全局时也能有效地运转。

尽管软件团队往往也是这样组织起来，但他们所致力的工作却常常没有同样的关注面切分及抽象层级。整洁的代码帮助我们在较低层的抽象层级上达成这一目标。本章将讨论如何在较高的抽象层级——系统 层级——上保持整洁。

11.2　将系统的构造与使用分开

首先，构造 与使用 是非常不一样的过程。当我走笔至此，投目窗外的芝加哥，看到有一间酒店正在建设。今天，那只是个框架结构，起重机和升降机附着在外面。忙碌的人们身穿工作服，头戴安全帽。大概一年之后，酒店就将建成。起重机和升降机都会消失无踪。建筑物变得整洁，覆盖着玻璃幕墙和漂亮的漆色。在其中工作和住宿的人，会看到完全不同的景象。

软件系统应将启始过程和启始过程之后的运行时逻辑分离开，在启始过程中构建应用对象，也会存在互相缠结的依赖关系。

每个应用程序都该留意 启始过程。那也是本章中我们首先要考虑的问题。将关注的方面分离开，是软件技艺中最古老也最重要的设计技巧。

不幸的是，多数应用程序都没有做分离处理。启始过程代码很特殊，被混杂到运行时逻辑中。下例就是典型的情形：

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl(...);  // Good enough default fo
```

```
r most cases?
  return service;
}
```

这就是所谓延迟初始化/赋值，也有一些好处。在真正用到对象之前，无需操心这种架空构造，启始时间也会更短，而且还能保证永远不会返回null值。

然而，我们也得到了MyServiceImpl及其构造器所需一切（我省略了那些代码）的硬编码依赖。不分解这些依赖关系就无法编译，即便在运行时永不使用这种类型的对象！

如果MyServiceImpl是个重型对象，则测试也会是个问题。我们必须确保在单元测试调用该方法之前，就给**service** 指派恰当的测试替身（TEST DOUBLE）[1] 或仿制对象（MOCK OBJECT）。由于构造逻辑与运行过程相混杂，我们必须测试所有的执行路径（例如，null值测试及其代码块）。有了这些权责，说明方法做了不止一件事，这样就略微违反了单一权责原则 。

最糟糕的大概是我们不知道MyServiceImpl在所有情形中是否都是正确的对象。我在代码注释中做了暗示。为什么该方法所属类必须知道全局情

景？我们是否真能知道在这里要用到的正确对象？是否真有可能存在一种放之四海而皆准的类型？

当然，仅出现一次的延迟初始化不算是严重问题。不过，在应用程序中往往有许多种类似的情况出现。于是，全局设置策略（如果有的话）在应用程序中四散分布，缺乏模块组织性，通常也会有许多重复代码。

如果我们勤于打造有着良好格式并且强固的系统，就不该让这类就手小技巧破坏模块组织性。对象构造的启始和设置过程也不例外。应当将这个过程从正常的运行时逻辑中分离出来，确保拥有解决主要依赖问题的全局性一贯策略。

11.2.1　分解main

将构造与使用分开的方法之一是将全部构造过程搬迁到main或被称之为main的模块中，设计系统的其余部分时，假设所有对象都已正确构造和设置（如图11-1所示）。

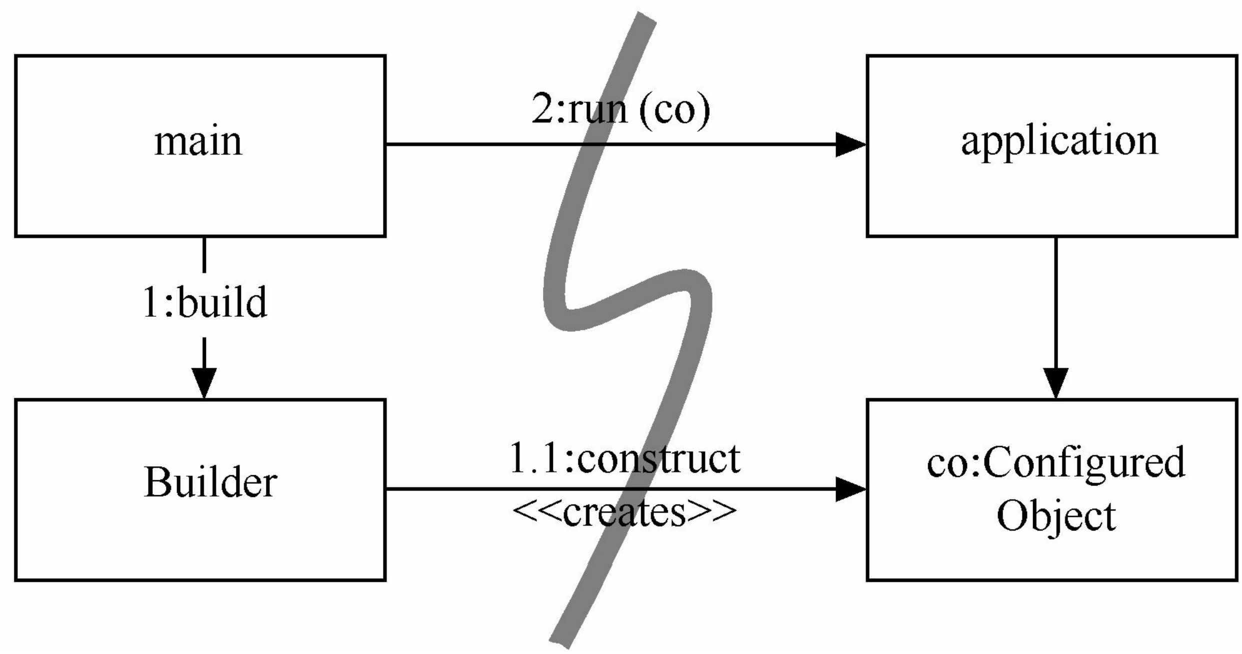


图11-1　将构造分解到main( )中

控制流程很容易理解。main函数创建系统所需的对象，再传递给应用程序，应用程序只管使用。注意看横贯main与应用程序之间隔篱的依赖箭头的方向。它们都从main函数向外走。这表示应用程序对main或者构造过程一无所知。它只是简单地指望一切已齐备。

11.2.2　工厂

当然，有时应用程序也要负责确定何时 创建对象。比如，在某个订单处理系统中，应用程序必须创建LineItem实体，添加到Order对象。在这种情况下，我们可以使用抽象工厂模式 [2] 让应用自行

控制何时创建LineItems，但构造的细节却隔离于应用程序代码之外。

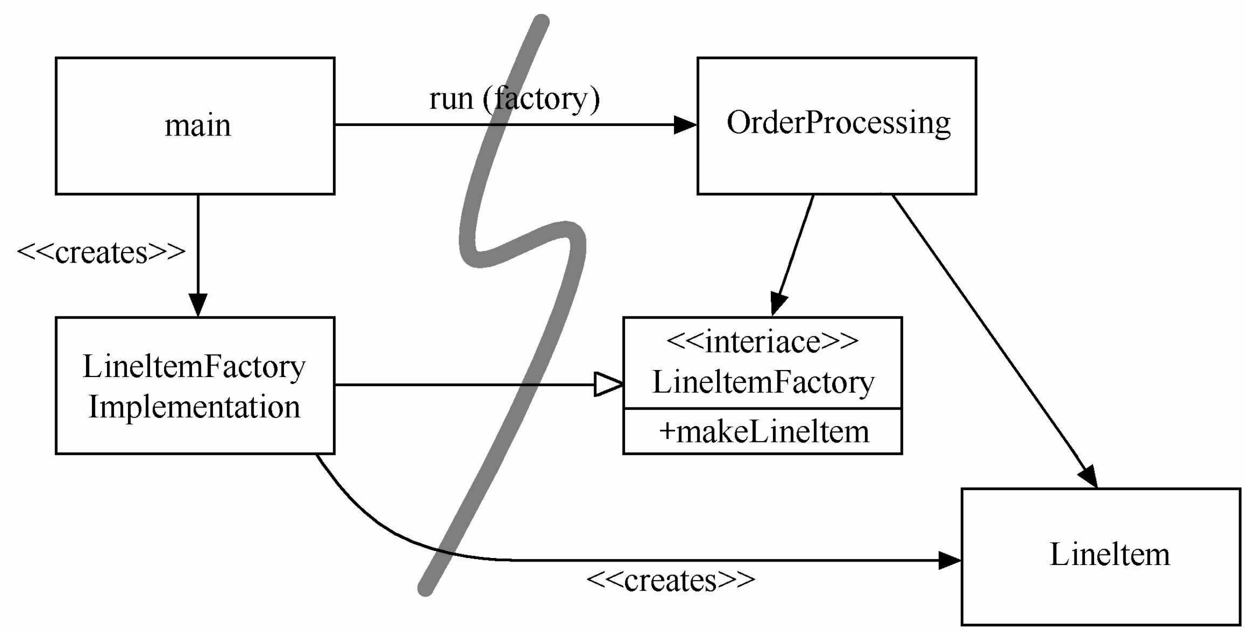


图11-2　使用工厂分离构造过程

再留意一下，所有依赖都是从main指向OrderProcessing应用程序。这代表应用程序与如何构建LineItem的细节是分离开来的。构建能力由LineItemFactoryImplementation持有，而LineItemFactoryImplementation又是在main这一边的。但应用程序能完全控制LineItem实体何时构建，甚至能传递应用特定的构造器参数。

11.2.3　依赖注入

有一种强大的机制可以实现分离构造与使用，那就是依赖注入（Dependency Injection，DI），控制反转（Inversion of Control，IoC）在依赖管理中

的一种应用手段[3]。控制反转将第二权责从对象中拿出来，转移到另一个专注于此的对象中，从而遵循了单一权责原则。在依赖管理情景中，对象不应负责实体化对自身的依赖。反之，它应当将这份权责移交给其他“有权力”的机制，从而实现控制的反转。因为初始设置是一种全局问题，这种授权机制通常要么是main例程，要么是有特定目的的容器。

JNDI查找是DI的一种“部分”实现。在JNDI中，对象请求目录服务器提供一种符合某个特定名称的“服务”。

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

调用对象并不控制真正返回对象的类别（当然前提是它实现了恰当的接口），但调用对象仍然主动分解了依赖。

真正的依赖注入还要更进一步。类并不直接分解其依赖，而是完全被动的。它提供可用于注入依赖的赋值器方法或构造器参数（或二者皆有）。在构造过程中，DI容器实体化需要的对象（通常按需创建），并使用构造器参数或赋值器方法将依赖

连接到一起。至于哪个依赖对象真正得到使用，是通过配置文件或在一个有特殊目的的构造模块中编程决定。

Spring框架提供了最有名的Java DI容器 [4] 。用户在XML配置文件中定义互相关联的对象，然后用Java代码请求特定的对象。稍后我们就会看到例子。

但延后初始化的好处是什么呢？这种手段在DI中也有其作用。首先，多数DI容器在需要对象之前并不构造对象。其次，许多这类容器提供调用工厂或构造代理的机制，而这种机制可为延迟赋值或类似的优化处理 所用 [5] 。

11.3 扩容

城市由城镇而来，城镇由聚居而来。一开始，道路狭窄，几乎无人涉足，随后逐渐拓宽。小型建筑和空地渐渐被更大的建筑所取代，一些地方最终矗立起摩天大楼。

一开始，供电、供水、下水、互联网（哇！）等服务全部欠奉。随着人口和建筑密度的增加，这些服务也开始出现。

这种成长并非全无阵痛。你有多少次开着车，艰难穿行过一个“道路改善”工程，问自己，“他们为什么不一开始就修条够宽的路呢？！”

不过那无论如何不可能实现。谁敢打包票说在一个小镇修建一条六车道的公路并不浪费呢？谁会想要这么一条穿过他们小镇的路呢？

“一开始就做对系统”纯属神话。反之，我们应该只去实现今天的用户故事，然后重构，明天再扩展系统、实现新的用户故事。这就是迭代和增量敏捷的精髓所在。测试驱动开发、重构以及它们打造出的整洁代码，在代码层面保证了这个过程的实现。

但在系统层面又如何？难道系统架构不需要预先做好计划吗？系统理所当然不可能从简单递增到复杂，它能行吗？

与物理系统相比软件系统比较独特。它们的架构都可以递增式地增长，只要我们持续将关注面恰当地切分。

如我们将见到的那样，软件系统短生命周期本质使这一切变得可行。我们先来看一个没有充分隔离关注问题的架构反例。

初始的EJB1和EJB2架构没有恰当地切分关注面，从而给有机增长压上了不必要的负担。比如一个持久Bank类的Entity Bean。Entity bean是关系数据在内存中的体现，换言之，是表格的一行。

首先，你要定义一个本地（进程内）或远程（分离的JVM）接口，供客户代码使用。代码清单11-1就是一种可能的本地接口：

代码清单11-1　Bank EJB的EJB2本地接口

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
```

```
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

上面列出了银行地址的几个属性，和一组该银行拥有的账户，其中每个账户的数据都由单独的Account EJB所持有。代码清单11-2展示了Bank bean的相应实现类。

代码清单11-2　相应的EJB2 Entity Bean实现

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
  // 业务逻辑...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
  public abstract void setState(String state);
  public abstract void setZipCode(String zip);
  public abstract Collection getAccounts();
  public abstract void setAccounts(Collection accounts);
```

```
  public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome = context.lookup("AccountHomeLo
cal");
    AccountLocal account = accountHome.create(accountDTO);
    Collection accounts = getAccounts();
    accounts.add(account);
  }
  // EJB container logic
  public abstract void setId(Integer id);
  public abstract Integer getId();
  public Integer ejbCreate(Integer id) { ... }
  public void ejbPostCreate(Integer id) { ... }
  // The rest had to be implemented but were usually empty:
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
}
```

我没有列出对应的LocalHome接口，该接口基本上是用来创建对象的，也没有列出你可能添加的Bank查找器（查询）。

最后，你要编写一个或多个XML部署说明，将对象相关映射细节指定给某个持久化存储空间，说明期望的事物行为、安全约束等。

业务逻辑与EJB2应用“容器”紧密耦合。你必须子类化容器类型，必须提供许多个该容器所需要的生命周期方法。

由于存在这种与重量级容器的紧耦合，隔离单元测试就很困难。有必要模拟出容器（这很难），或者花费大量时间在真实服务器上部署EJB和测试。也由于耦合的存在，在EJB2架构之外的复用实际上变得不可能。

最终，连面向对象编程本身也被侵蚀。bean不能继承自另一个bean。留意添加新账号的逻辑。在EJB2 bean中，定义一种本质上是无行为struct的“数据传输对象”（DTO）很常见。这往往会导致拥有同样数据的冗余类型出现，而且也需要在对象之间复制数据的八股式代码。

横贯式关注面

在某些领域，EBJ2架构已经很接近于真正的关注面切分。例如，在与源代码分离的部署描述中声明了期待的事务、安全及部分持久化行为。

注意，持久化之类关注面 倾向于横贯某个领域的天然对象边界。你会想用同样的策略来持久化所有对象，例如，使用DBMS [6] 而非平面文件，表名和列名遵循某种命名约定，采用一致的事务语义，等等。

原则上，你可以从模块、封装的角度推理持久

化策略。但在实践上，你却不得不将实现了持久化策略的代码铺展到许多对象中。我们用术语“横贯式关注面 ”来形容这类情况。同样，持久化框架和领域逻辑，孤立地看也可以是模块化的。问题在于横贯 这些领域的情形。

实际上，EJB架构处理持久化、安全和事务的方法要早于面向方面编程（aspect-oriented programming， AOP）[7]，而AOP是一种恢复横贯式关注面模块化的普适手段。

在AOP中，被称为方面 （aspect）的模块构造指明了系统中哪些点的行为会以某种一致的方式被修改，从而支持某种特定的场景。这种说明是用某种简洁的声明或编程机制来实现的。

以持久化为例，可以声明哪些对象和属性（或其模式）应当被持久化，然后将持久化任务委托给持久化框架。行为的修改由AOP框架以无损方式[8]在目标代码中进行。下面来看看Java中的三种方面或类似方面的机制。

11.4　Java代理

Java代理适用于简单的情况，例如在单独的对象或类中包装方法调用。然而，JDK提供的动态代理仅能与接口协同工作。对于代理类，你得使用字节码操作库，比如CGLIB、ASM或Javassist [9] 。

代码清单11-3展示了为我们的Bank应用程序提供持久化支持的JDK代理，代码仅覆盖设置和取得账号列表的方法。

代码清单11-3　JDK代理范例

```
// Bank.java (suppressing package names...)
import java.utils.*;

// The abstraction of a bank.
public interface Bank {
  Collection<Account> getAccounts();
  void setAccounts(Collection<Account> accounts);
}

// BankImpl.java
import java.utils.*;

// The "Plain Old Java Object" (POJO) implementing the abstract
ion.
public class BankImpl implements Bank {
  private List<Account> accounts;

  public Collection<Account> getAccounts() {
    return accounts;
  }
```

```
  public void setAccounts(Collection<Account> accounts) {
    this.accounts = new ArrayList<Account>();
    for (Account account: accounts) {
      this.accounts.add(account);
    }
  }
}

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// "InvocationHandler" required by the proxy API.
public class BankProxyHandler implements InvocationHandler {
  private Bank bank;

  public BankHandler (Bank bank) {
    this.bank = bank;
  }

  // Method defined in InvocationHandler
  public Object invoke(Object proxy, Method method, Object[] arg
s)
    throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getAccounts")) {
      bank.setAccounts(getAccountsFromDatabase());
      return bank.getAccounts();
    } else if (methodName.equals("setAccounts")) {
      bank.setAccounts((Collection<Account>) args[0]);
      setAccountsToDatabase(bank.getAccounts());
      return null;
    } else {
      ...
    }
  }
  // Lots of details here:
  protected Collection<Account> getAccountsFromDatabase() { ...
}
  protected void setAccountsToDatabase(Collection<Account> accou
nts) { ... }
}

// Somewhere else...
```

```
Bank bank = (Bank) Proxy.newProxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl()));
```

我们定义了将被代理包装 起来的接口Bank，还有旧式的Java对象（Plain-Old Java Object，POJO）BankImpl，该对象实现业务逻辑（稍后再来看POJO）。

Proxy API需要一个InvocationHandler对象，用来实现对代理的全部Bank方法调用。BankProxyHandler使用Java反射API将一般方法调用映射到BankImpl中的对应方法，以此类推。

即便对于这样简单的例子，也有许多相对复杂的代码 [10] 。使用那些字节操作类库也同样具有挑战性。代码量和复杂度是代理的两大弱点，创建整洁代码变得很难！另外，代理也没有提供在系统范围内指定执行点的机制，而那正是真正的AOP解决方案所必须的 [11] 。

11.5　纯Java AOP框架

幸运的是，编程工具能自动处理大多数代理模板代码。在数个Java框架中，代理都是内嵌的，如Spring AOP和JBoss AOP等，从而能够以纯Java代码实现面向方面编程[12]。在Spring中，你将业务逻辑编码为旧式**Java**对象。POJO自扫门前雪，并不依赖于企业框架（或其他域）。因此，它在概念上更简单、更易于测试驱动。相对简单性也较易于保证正确地实现相应的用户故事，并为未来的用户故事维护和改进代码。

使用描述性配置文件或API，你把需要的应用程序构架组合起来，包括持久化、事务、安全、缓存、恢复等横贯性问题。在许多情况下，你实际上只是指定Spring或Jboss类库，框架以对用户透明的方式处理使用Java代理或字节代码库的机制。这些声明驱动了依赖注入（DI）容器，DI容器再实体化主要对象，并按需将对象连接起来。

代码清单11-4展示了Spring V2.5配置文件app.xml的典型片段[13]。

代码清单11-4　Spring 2.x的配置文件

```
<beans>
...
<bean id="appDataSource"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close"
p:driverClassName="com.mysql.jdbc.Driver"
p:url="jdbc:mysql://localhost:3306/mydb"
p:username="me"/>

<bean id="bankDataAccessObject"
class="com.example.banking.persistence.BankDataAccessObject"
p:dataSource-ref="appDataSource"/>

<bean  id="bank"
class="com.example.banking.model.Bank"
p:dataAccessObject-ref="bankDataAccessObject"/>
...
</beans>
```

每个bean就像是嵌套“俄罗斯套娃”中的一个，每个由数据存取器对象（DAO）代理（包装）的Bank都有个域对象，而bean本身又是由JDBC驱动程序数据源代理（如图11-3所示）。

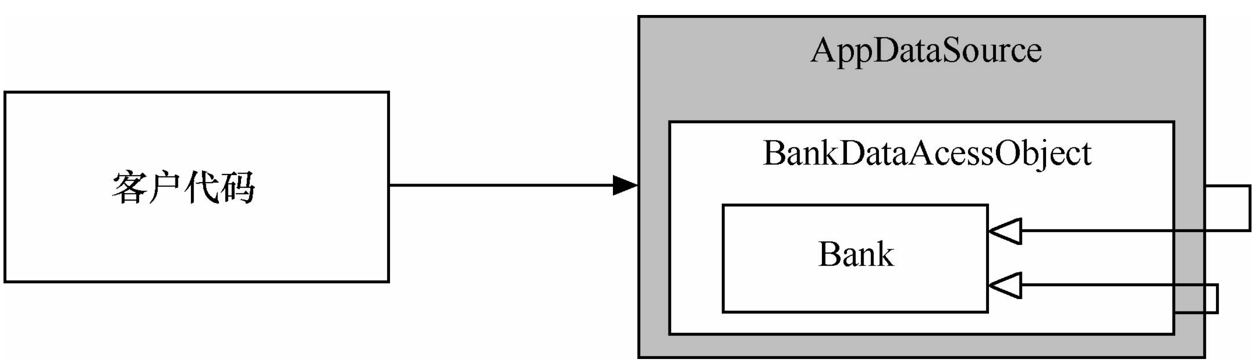


图11-3　“俄罗斯套娃”式的油漆工模式

客户代码以为调用的是Bank对象的getAccount()方法，其实它是在与一组扩展Bank POJO基础行为的油漆工（DECORATOR）[14]对象中最外面的那个沟通。

在应用程序中，只添加了少数几行代码，用来向DI容器请求系统中的顶层对象，如XML文件中所定义的那样。

```
XmlBeanFactory bf =
    new XmlBeanFactory(new ClassPathResource("app.xml", getClass
()));
Bank bank = (Bank) bf.getBean("bank");
```

只有区区几行与Spring相关的Java代码，应用程序几乎完全与Spring分离，消除了EJB2之类系统中那种紧耦合问题。

尽管XML可能会冗长且难以阅读[15]，配置文件中定义的“策略”还是要比那种隐藏在幕后自动创建的复杂的代理和方面逻辑来得简单。这种类型的架构是如此引人注目，Spring之类的框架最终导致了EJB标准在第3版的彻底变化。使用XML配置文件和/或Java 5 annotation，EJB3很大程度上遵循了Spring通过描述性手段支持横贯式关注面的模型。

代码清单11-5展示了用EJB3重写的Bank对象[16]。

代码清单11-5　EJB3版本的Bank

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private int id;
  @Embeddable // An object "inlined" in Bank’s DB row
  public class Address {
    protected String streetAddr1;
    protected String streetAddr2;
    protected String city;
    protected String state;
    protected String zipCode;
  }

  @Embedded
  private Address address;

  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,

              mappedBy="bank")
  private Collection<Account> accounts = new ArrayList<Account>(
);

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public void addAccount(Account account) {
```

```
    account.setBank(this);
    accounts.add(account);
  }

  public Collection<Account> getAccounts() {
    return accounts;
  }

  public void setAccounts(Collection<Account> accounts) {
    this.accounts = accounts;
  }
}
```

上列代码要比原本的EJB2代码整洁多了。有些实体细节仍然在annotation中存在。不过，因为没有任何信息超出annotation之外，代码依然整洁、清晰，也因此而易于测试驱动、易于维护。

如果愿意的话，annotation中有些或全部持久化信息可以转移到XML部署描述中，只留下真正的纯POJO。如果持久化映射细节不会频繁改动，许多团队可能会选择保留annotation，但与EJB2那种侵害性相比还是少了很多问题。

11.6　AspectJ的方面

通过方面来实现关注面切分的功能最全的工具是AspectJ语言[17]，一种提供“一流的”将方面作为模块构造处理支持的Java扩展。在80%~90%用到方面特性的情况下，Spring AOP和JBoss AOP提供的纯Java实现手段足够使用。然而，AspectJ却提供了一套用以切分关注面的丰富而强有力的工具。AspectJ的弱势在于，需要采用几种新工具，学习新语言构造和使用方式。

藉由AspectJ近期引入的“annotation form”（使用Java 5 annotation定义纯Java代码的方面），新工具采用的问题大大减少。另外，Spring Framework也有一些让拥有较少AspectJ经验的团队更容易组合基于annotation的方面的特性。

关于AspectJ的全面探讨已经超出本书范围。更多信息可参见[AspectJ]、[Colyer]和[Spring]。

11.7　测试驱动系统架构

通过方面式的手段切分关注面的威力不可低估。假使你能用POJO编写应用程序的领域逻辑，在代码层面与架构关注面分离开，就有可能真正地用测试来驱动 架构。采用一些新技术，就能将架构按需从简单演化到精细。没必要先做大设计（Big Design Up Front，BDUF）[18]。实际上，BDUF甚至是有害的，它阻碍改进，因为心理上会抵制丢弃既成之事，也因为架构上的方案选择影响到后续的设计思路。

建筑设计师不得不做BDUF，因为一旦建造过程开始，就不可能对大型物理建筑的结构做根本性改动[19]。尽管软件也有物理[20]的一面，只要软件的构架有效切分了各个关注面，还是有可能做根本性改动的。

这意味着我们可以从“简单自然”但切分良好的架构开始做软件项目，快速交付可工作的用户故事，随着规模的增长添加更多基础架构。有些世界上最大的网站采用了精密的数据缓存、安全、虚拟化等技术，获得了极高的可用性和性能，在每个抽象层和范围之内，那些最小化耦合的设计都简单到

位，效率和灵活性也随之而来。

当然，这不是说要毫无准备地进入一个项目。对于总的覆盖范围、目标、项目进度和最终系统的总体构架，我们会有所预期。不过，我们必须有能力随机应变。

EJB早期架构就是一种著名的过度工程化而没能有效切分关注面的API。在没能真正得到使用时，设计得再好的API也等于是杀鸡用牛刀。优秀的API在大多数时间都该在视线之外，这样团队才能将创造力集中在要实现的用户故事上。否则，架构上的约束就会妨碍向客户交付优化价值的软件。

概言之，

最佳的系统架构由模块化的关注面领域组成，每个关注面均用纯Java（或其他语言）对象实现。不同的领域之间用最不具有侵害性的方面或类方面工具整合起来。这种架构能测试驱动，就像代码一样。

11.8　优化决策

模块化和关注面切分成就了分散化管理和决策。在巨大的系统中，不管是一座城市或一个软件项目，无人能做所有决策。

众所周知，最好是授权给最有资格的人。但我们常常忘记了，延迟决策至最后一刻 也是好手段。这不是懒惰或不负责；它让我们能够基于最有可能的信息做出选择。提前决策是一种预备知识不足的决策。如果决策太早，就会缺少太多客户反馈、关于项目的思考和实施经验。

拥有模块化关注面的POJO系统提供的敏捷能力，允许我们基于最新的知识做出优化的、时机刚好的决策。决策的复杂性也降低了。

11.9 明智使用添加了可论证价值的标准

建筑构造大有可观，既因为新建筑的构建过程（即便是在隆冬季节），也因为那些现今科技所能实现的超凡设计。建筑业是一个成熟行业，有着高度优化的部件、方法和久经岁月历练的标准。

即便是轻量级和更直截了当的设计已足敷使用，许多团队还是采用了EJB2架构，只因为EJB2是个标准。我见过一些团队，纠缠于这个或那个名声大噪 的标准，却丧失了对为客户实现价值的关注。

有了标准，就更易复用想法和组件、雇用拥有相关经验的人才、封装好点子，以及将组件连接起来。不过，创立标准的过程有时却漫长到行业等不及的程度，有些标准没能与它要服务的采用者的真实需求相结合。

11.10　系统需要领域特定语言

建筑，与大多数其他领域一样，发展出一套丰富的语言，有词汇、熟语和清晰而简洁地表达基础信息的句式[21]。在软件领域，领域特定语言（Domain-Specific Language，DSL）[22]最近重受关注。DSL是一种单独的小型脚本语言或以标准语言写就的API，领域专家可以用它编写读起来像是组织严谨的散文一般的代码。

优秀的DSL填平了领域概念和实现领域概念的代码之间的“壕沟”，就像敏捷实践优化了开发团队和甲方之间的沟通一样。如果你用与领域专家使用的同一种语言来实现领域逻辑，就会降低不正确地将领域翻译为实现的风险。

DSL在有效使用时能提升代码惯用法和设计模式之上的抽象层次。它允许开发者在恰当的抽象层级上直指代码的初衷。

领域特定语言允许所有抽象层级和应用程序中的所有领域，从高级策略到底层细节，使用POJO来表达。

11.11　小结

　　系统也应该是整洁的。侵害性架构会湮灭领域逻辑，冲击敏捷能力。当领域逻辑受到困扰，质量也就堪忧，因为缺陷更易隐藏，用户故事更难实现。当敏捷能力受到损害时，生产力也会降低，TDD的好处遗失殆尽。

　　在所有的抽象层级上，意图都应该清晰可辨。只有在编写POJO并使用类方面的机制来无损地组合其他关注面时，这种事情才会发生。

　　无论是设计系统或单独的模块，别忘了使用大概可工作的最简单方案 。

11.12　文献

[Alexander]： Christopher Alexander, *A Timeless Way of Building*, Oxford University Press, New York, 1979.

[AOSD]： Aspect-Oriented Software Development port, http://aosd.net

[ASM]： ASM Home Page, http://asm.objectweb.org/

[AspectJ]： http://eclipse.org/aspectj

[CGLIB]： Code Generation Library, http://cglib.sourceforge.net/

[Colyer]： Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, *Eclipse AspectJ*, Person Education, Inc., Upper Saddle River, NJ, 2005.

[DSL]： Domain-specific programming language, http://en.wikipedia.org/wiki/Domain-specific_programming_language

[Fowler]： Inversion of Control Containers and the Dependency Injection pattern, http://martinfowler.com/articles/injection.html

[Goetz]： Brian Goetz, *Java Theory and Practice: Decorating with Dynamic Proxie* s, http://www.ibm.com/developerworks/java/library/j-jtp08305.html

[Javassist]： Javassist Home Page, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[JBoss]： JBoss Home Page, http://jboss.org

[JMock]： JMock—A Lightweight Mock Object Library for Java, http://jmock.org

[Kolence]： Kenneth W. Kolence, Software physics and computer performance measure- ments, *Proceedings of the ACM annual conference—Volume 2* , Boston, Massachusetts, pp. 1024–1040, 1972.

[Spring]： *The Spring Framework* , http://www.springframework.org

[Mezzaros07]： *XUnit Patterns* , Gerard Mezzaros, Addison-Wesley, 2007.

[GOF]： *Design Patterns: Elements of Reusable Object Oriented Software* , Gamma et al., Addison-Wesley, 1996.

---

[1]　原注：[Mezzaros07]。

[2]　原注：[GOF]。

[3]　原注：可参见[Fowler]。

[4]　原注：见[Spring]。另外也有一个Spring.NET框架。

[5]　原注：别忘记延迟初始化/赋值只是一种优化手段，而且可能是一种不成熟的手段。

[6]　原注：数据管理系统。

[7]　原注：查阅[AOSD]获取有关方面的一般信息，查阅[AspectJ]和[Colyer]获取有关AspectJ的信息。

[8]　原注：即无需手工修改源代码。

[9]　原注：见[CGLIB]、[ASM]和[Javassist]。

[10]　原注：要想了解更多关于Proxy API及其用法，请参阅[Goetz]。

[11]　原注：AOP有时会与实现它的技术相混淆，例如方法拦截和通过代理做的“封包”。AOP系统的真正价值在于用简洁和模块化的方式指定系统行为。

[12]　原注：见[Spring]和[JBoss]。“纯Java”表示不使用AspectJ。

[13]　原注：摘自 http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25 。

[14]　原注：[GOF]。

[15]　原注：可以使用遵循“约定胜于配置”的机制和Java 5 annotation来减少外露的连接逻辑，从而简化这个例子。

[16]　原注：摘自 http://www.onjava.com/pub/a/onjava/2006/05/17/standa with-ejb3-java-persistence-api.html 。

[17]　原注：参见[AspectJ]和[Colyer]。

[18]　原注：BDUF是一种预先设计好一切实现的方式，不能与先做设计（up-front design）的良好实践手段相混淆。

[19]　原注：即便在构建开始之后，也会有大量迭代式的考察和细节讨论。

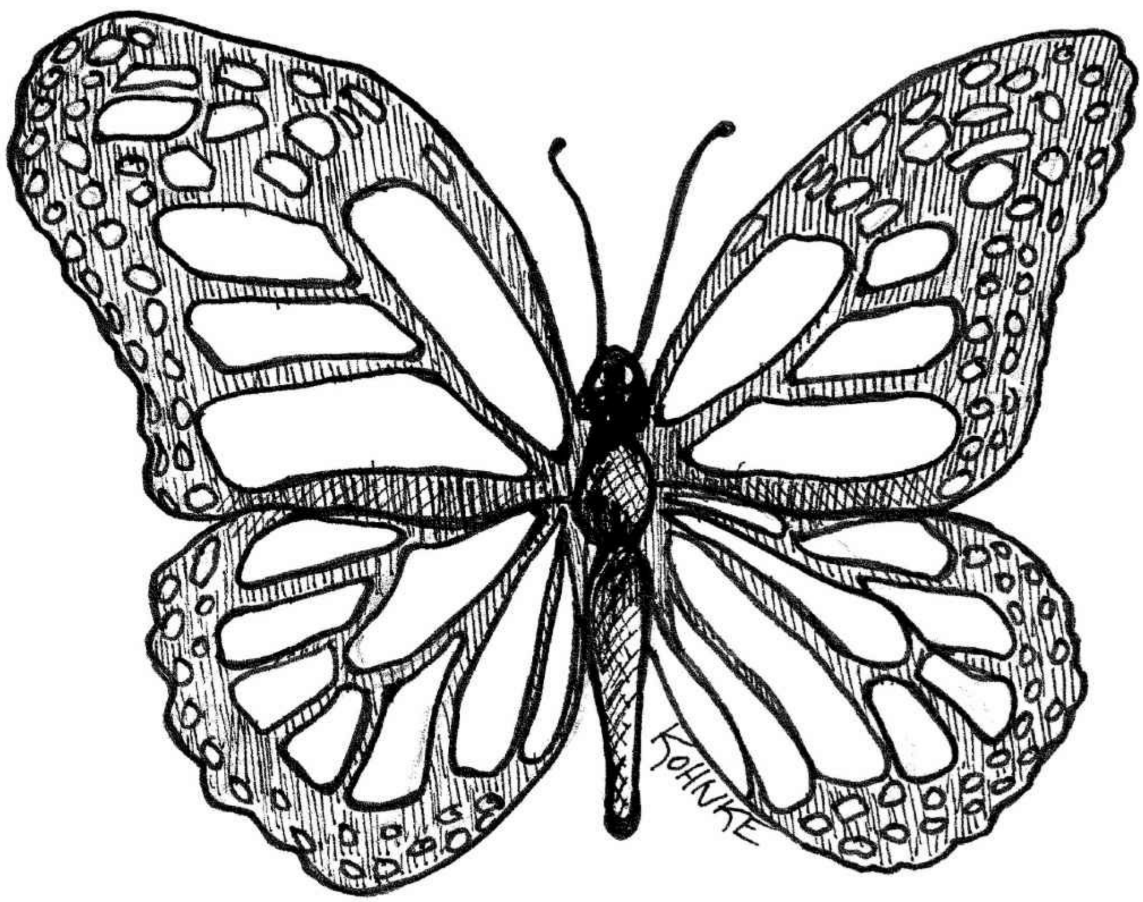
[20]　原注：“软件物理”一词最早由[Kolence]提出。

[21]　原注：[Alexander]的著作对软件社区影响至深。

[22]　原注：参见[DSL]。[JMock]是创建DSL的Java API的优秀范例。

第12章　迭进

Jeff Langr



12.1　通过迭进设计达到整洁目的

假使有4条简单的规矩，跟着做就能帮助你创建优良的设计，会如何？假使遵循这些规矩你就能洞见代码的结构和设计，更轻易地应用SRP和DIP之类原则，又会如何？假使这4条规则有利于良好的设计“浮现”出来，又会如何？

我们中的许多人认为，Kent Beck关于简单设计 [1] 的四条规则，对于创建具有良好设计的软件有着莫大的帮助。

- 据Kent所述，只要遵循以下规则，设计就能变得“简单”：
- 运行所有测试；
- 不可重复；
- 表达了程序员的意图；
- 尽可能减少类和方法的数量；
- 以上规则按其重要程度排列。

12.2　简单设计规则1：运行所有测试

设计必须制造出如预期一般工作的系统，这是首要因素。系统也许有一套绝佳设计，但如果缺乏验证系统是否真按预期那样工作的简单方法，那就无异于纸上谈兵。

全面测试并持续通过所有测试的系统，就是可测试的系统。看似浅显，但却重要。不可测试的系统同样不可验证。不可验证的系统，绝不应部署。

幸运的是，只要系统可测试，就会导向保持类短小且目的单一的设计方案。遵循SRP的类，测试起来较为简单。测试编写得越多，就越能持续走向编写较易测试的代码。所以，确保系统完全可测试能帮助我们创建更好的设计。

紧耦合的代码难以编写测试。同样，编写测试越多，就越会遵循DIP之类规则，使用依赖注入、接口和抽象等工具尽可能减少耦合。如此一来，设计就有长足进步。

遵循有关编写测试并持续运行测试的简单、明

确的规则，系统就会更贴近OO低耦合度、高内聚度的目标。编写测试引致更好的设计。

12.3　简单设计规则2～4：重构

有了测试，就能保持代码和类的整洁，方法就是递增式地重构代码。添加了几行代码后，就要暂停，琢磨一下变化了的设计。设计退步了吗？如果是，就要清理它，并且运行测试，保证没有破坏任何东西。测试消除了对清理代码就会破坏代码的恐惧 。

在重构过程中，可以应用有关优秀软件设计的一切知识。提升内聚性，降低耦合度，切分关注面，模块化系统性关注面，缩小函数和类的尺寸，选用更好的名称，如此等等。这也是应用简单设计后三条规则的地方：消除重复，保证表达力，尽可能减少类和方法的数量。

12.4　不可重复

重复是拥有良好设计系统的大敌。它代表着额外的工作、额外的风险和额外且不必要的复杂度。重复有多种表现。极其雷同的代码行当然是重复。类似的代码往往可以调整得更相似，这样就能更容易地进行重构。重复也有实现上的重复等其他一些形态。例如，在某个群集类中可能会有两个方法：

```
int size()  {}
boolean isEmpty()  {}
```

这两个方法可以分别实现。isEmpty方法跟踪一个布尔值，而size方法则跟踪一个计数器。或者，也可以通过在isEmpty中使用size方法来消除重复：

```
boolean isEmpty() {
  return 0 == size();
}
```

要想创建整洁的系统，需要有消除重复的意愿，即便对于短短几行也是如此。例如以下代码：

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThresho
ld)
    return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01
f);

  RenderedOp newImage = ImageUtilities.getScaledImage(
      image, scalingFactor, scalingFactor);
  image.dispose();
  System.gc();
  image = newImage;
}
public synchronized void rotate(int degrees) {
  RenderedOp newImage = ImageUtilities.getRotatedImage(
      image, degrees);
  image.dispose();
  System.gc();
  image = newImage;
}
```

要保持系统整洁，应该消除scaleToOneDimension和rotate方法里面的少量重复：

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThresho
ld)
    return;
  float scalingFactor = desiredDimension / imageDimension;
  scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01
f);
  replaceImage(ImageUtilities.getScaledImage(
```

```
     image, scalingFactor, scalingFactor));
}
public synchronized void rotate(int degrees) {
  replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}
private void replaceImage(RenderedOp newImage) {
  image.dispose();
  System.gc();
  image = newImage;
}
```

做了一点点共性抽取后，我们意识到已经违反了SRP原则。所以，可以把一个新方法分解到另外的类中，从而提升其可见性。团队中的其他成员也许会发现进一步抽象新方法的机会，并且在其他场

景中复用之。“小规模复用”可大量降低系统复杂性。要想实现大规模复用，必须理解如何实现小规模复用。

模板方法模式 [2] 是一种移除高层级重复的通用技巧。例如：

```
public class VacationPolicy {
  public void accrueUSDivisionVacation() {
    // code to calculate vacation based on hours worked to date
    // ...
    // code to ensure vacation meets US minimums
    // ...
    // code to apply vaction to payroll record
    // ...
  }
  public void accrueEUDivisionVacation() {
    // code to calculate vacation based on hours worked to date
    // ...
    // code to ensure vacation meets EU minimums
    // ...
    // code to apply vaction to payroll record
    // ...
  }
}
```

除了计算法定最少数量假期的部分，accrueUSDivisionVacation和accrueEuropeanDivisionVacation中有大量代码雷同。那部分的算法，依据员工类型而变。

可以通过应用模板方法模式来消除明显的重

复。

```
abstract public class VacationPolicy {
  public void accrueVacation() {
    calculateBaseVacationHours();



    alterForLegalMinimums();



    applyToPayroll();



  }

  private void calculateBaseVacationHours() { /* ... */ };
  abstract protected void alterForLegalMinimums();
  private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // US specific logic
  }
}

public class EUVacationPolicy extends VacationPolicy {
  @Override protected void alterForLegalMinimums() {
    // EU specific logic
  }
}
```

子类填充了accrueVacation算法中的“空洞”，提供不重复的信息。

12.5　表达力

我们中的大多数人都经历过费解代码的纠缠。我们中的许多人自己就编写过费解的代码。写出自己 能理解的代码很容易，因为在写这些代码时，我们正深入于要解决的问题中。代码的其他维护者不会那么深入，也就不易理解代码。

软件项目的主要成本在于长期维护。为了在修改时尽量降低出现缺陷的可能性，很有必要理解系统是做什么的。当系统变得越来越复杂，开发者就需要越来越多的时间来理解它，而且也极有可能误解。所以，代码应当清晰地表达其作者的意图。作者把代码写得越清晰，其他人花在理解代码上的时间也就越少，从而减少缺陷，缩减维护成本。

可以通过选用好名称来表达。我们想要听到好类名和好函数名，而且在查看其权责时不会大吃一惊。

也可以通过保持函数和类尺寸短小来表达。短小的类和函数通常易于命名，易于编写，易于理解。

还可以通过采用标准命名法来表达。例如，设

计模式很大程度上就关乎沟通和表达。通过在实现这些模式的类的名称中采用标准模式名，例如COMMAND或VISITOR，就能充分地向其他开发者描述你的设计。

编写良好的单元测试也具有表达性。测试的主要目的之一就是通过实例起到文档的作用。读到测试的人应该能很快理解某个类是做什么的。

不过，做到有表达力的最重要方式却是尝试。有太多时候，我们写出能工作的代码，就转移到下一个问题上，没有下足功夫调整代码，让后来者易于阅读。记住，下一位读代码的人最有可能是你自己。

所以，多少尊重一下你的手艺吧。花一点点时间在每个函数和类上。选用较好的名称，将大函数切分为小函数，时时照拂自己创建的东西。用心是最珍贵的资源。

12.6　尽可能少的类和方法

即便是消除重复、代码表达力和SRP等最基础的概念也会被过度使用。为了保持类和函数短小，我们可能会造出太多的细小类和方法。所以这条规则也主张函数和类的数量要少。

类和方法的数量太多，有时是由毫无意义的教条主义导致的。例如，某个编码标准就坚称应当为每个类创建接口。也有开发者认为，字段和行为必须切分到数据类和行为类中。应该抵制这类教条，采用更实用的手段。

我们的目标是在保持函数和类短小的同时，保持整个系统短小精悍。不过要记住，这在关于简单设计的四条规则里面是优先级最低的一条。所以，尽管使类和函数的数量尽量少是很重要的，但更重要的却是测试、消除重复和表达力。

12.7　小结

有没有能替代经验的一套简单实践手段呢？当然不会有。另一方面，本章中写到的实践来自于本书作者数十年经验的精练总结。遵循简单设计的实践手段，开发者不必经年学习就能掌握好的原则和模式。

12.8 文献

[XPE]：*Extreme Programming Explained: Embrace Change* , Kent Beck,Addison- Wesley, 1999.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software* , Gamma et al., Addison-Wesley, 1996.

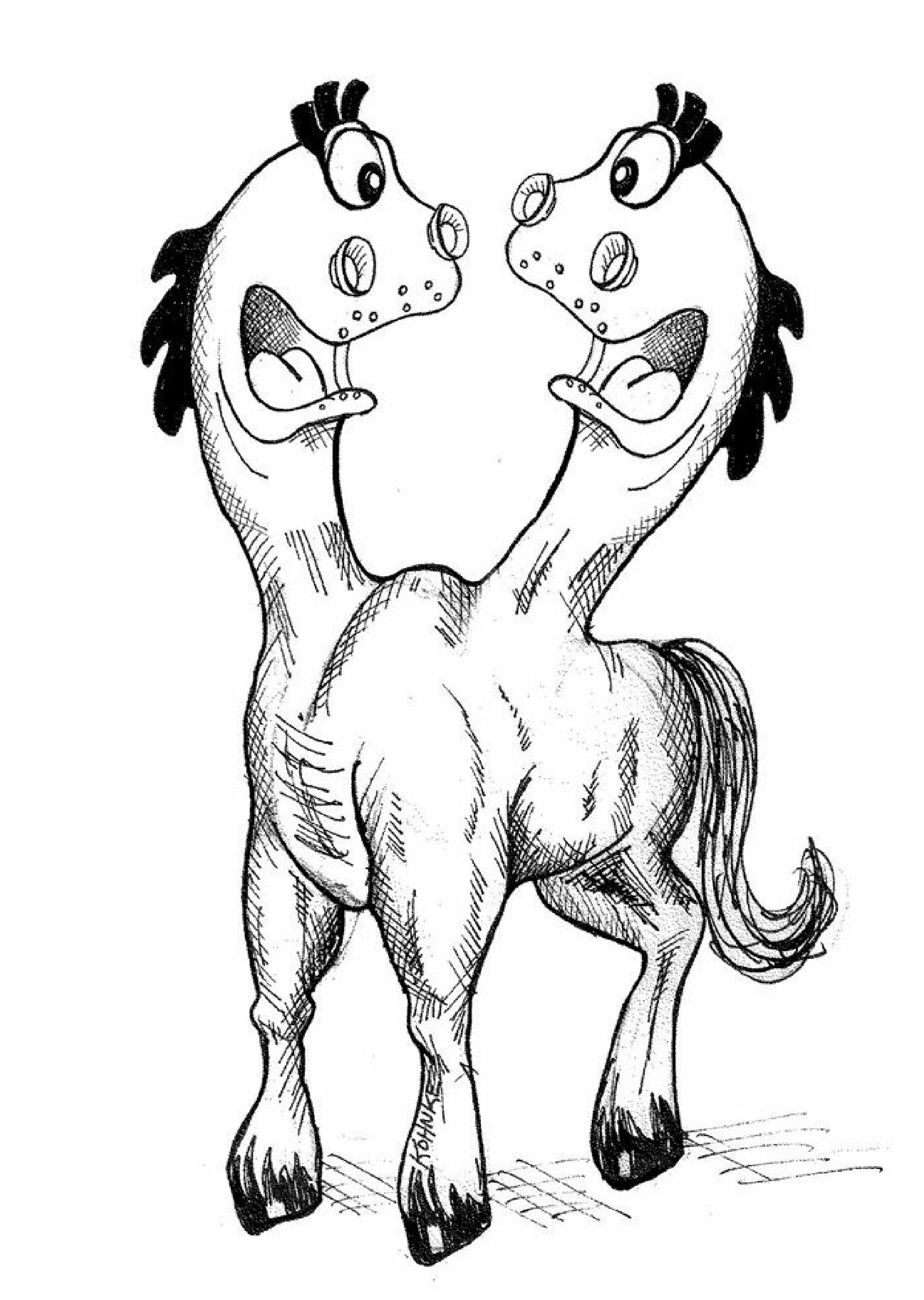
---

[1] 原注：[XPE]。

[2] 原注：[GOF]。

第**13**章　并发编程

Brett L.Schuchert



“对象是过程的抽象。线程是调度的抽象。”

——James O Coplien [1]

编写整洁的并发程序很难——非常难。编写在单线程中执行的代码简单得多。编写表面上看来不错、深入进去却支离破碎的多线程代码也简单。系统一旦遭受压力，这种代码就扛不住了。

本章将讨论并发编程的需求及其困难之处，并给出一些对付这些难点、编写整洁的并发代码的建议。最后，我们将讨论与测试并发代码有关的问题。

整洁的并发编程是个复杂话题，值得用一整本书来讨论。本书只做概览，并在“并发编程II”一章中提供更详细的指引。如果你只是对并发好奇，阅读本章就足够了。如果你需要更深入地理解并发，就应读完整个指引章节。

13.1　为什么要并发

并发是一种解耦策略。它帮助我们把做什么（目的）和何时（时机）做分解开。在单线程应用中，目的与时机紧密耦合，很多时候只要查看堆栈追踪即可断定应用程序的状态。调试这种系统的程序员可以设定断点或者断点序列，通过查看到达哪个断点来了解系统状态。

解耦目的与时机能明显地改进应用程序的吞吐量和结构。从结构的角度来看，应用程序看起来更像是许多台协同工作的计算机，而不是一个大循环。系统因此会更易于被理解，给出了许多切分关注面的有力手段。

例如，Web应用的Servlet标准模式。这类系统运行于Web或EJB容器的保护伞之下，Web或EJB为你部分地处理并发问题。当有Web请求时，servlet就会异步执行。Servlet程序员无需管理所有的请求。原则上，每次servlet是在自己的小世界中执行，与其他servlet的执行是分离的。

当然，如果只是那么简单，也就没必要写这一章了。实际上，Web容器提供的解耦手段离完美还差得远。Servlet程序员得非常警惕、非常小心地保

证并发程序不出错。同样，servlet模式的结构性好处还是很明显。

但结构并非采用并发的唯一动机。有些系统对响应时间和吞吐量有要求，需要手工编写并发解决方案。例如，考虑一个单线程信息聚合程序，它从许多Web站点获取信息，再合并写入日志中。因为该系统是单线程的，它会逐个访问Web站点，在开始下一个之前等待当前站点访问完毕。每天的执行时间必须少于24个小时。然而，随着要访问的站点越来越多，采集所有数据花费的时间也越来越多，最终超过了24个小时的限制。单线程程序许多时间花在等待Web套接字I/O结束上面。通过采用同时访问多个站点的多线程算法，就能改进性能。

或者，考虑某个每次花费1秒钟处理一个用户请求的系统。该系统在用户量较少的时候响应及时，但随着用户数增加，系统的响应时间也增加了。没人想排在150个人后面!通过并发处理多个用户请求，就能改进系统响应时间。

再或者，考虑某个解释大量数据集、但只在处理完全部数据后给出一个完整解决方案的系统。或许可以在独立的计算机上处理每个数据集，那样的话许多数据集就能并行地得到处理。

迷思与误解

看来有足够的理由采用并发方案。然而，如前文所述，并发编程很难 。如果你不那么细心，就会搞出不堪入目的东西来。看看以下常见的迷思和误解：

（1）并发总能改进性能

并发有时 能改进性能，但只在多个线程或处理器之间能分享大量等待时间的时候管用。事情没那么简单。

（2）编写并发程序无需修改设计

事实上，并发算法的设计有可能与单线程系统的设计极不相同。目的 与时机 的解耦往往对系统结构产生巨大影响。

（3）在采用Web或EJB容器的时候，理解并发问题并不重要

实际上，你最好了解容器在做什么，了解如何对付本章后文将提到的并发更新、死锁等问题。

下面是一些有关编写并发软件的中肯说法：

- 并发会在性能和编写额外代码上增加一些开销；
- 正确的并发是复杂的，即便对于简单的问题也是如此；
- 并发缺陷并非总能重现，所以常被看做偶发事件[2]而忽略，未被当做真的缺陷看待；
- 并发常常需要对设计策略的根本性修改。

13.2 挑战

并发编程为何如此之难？来看看下面这个小型类：

```
public class X {
  private int lastIdUsed;
  public int getNextId() {
       return ++lastIdUsed;
  }
}
```

比如，创建x的一个实体，将lastIdUsed设置为42，在两个线程中共享这个实体。假设这两个线程都调用getNextId( )方法，结果可能有三种输出：

- 线程一得到值43，线程二得到值44，lastIdUsed为44；
- 线程一得到值44，线程二得到值43，lastIdUsed为44；
- 线程一得到值43，线程二得到值43，lastIdUsed为43。

第三种结果令人惊异[3]，当两个线程相互影响时就会出现这种情况。这是因为线程在执行那行Java代码时有许多可能路径可行，有些路径会产生

错误的结果。有多少种不同路径呢？要真正回答这个问题，需要理解Just-In-Time编译器如何对待生成的字节码，还要理解Java内存模型认为什么东西具有原子性。

简答一下，就生成的字节码而言，对于在getNextId方法中执行的那两个线程，有12870种不同的可能执行路径[4]。如果lastIdUsed的类型从int变为long，则可能路径的数量将增至2704156种。当然，多数路径都得到正确结果。问题是其中一些不能得到正确结果 。

13.3　并发防御原则

下面给出一系列防御并发代码问题的原则和技巧。

13.3.1　单一权责原则

单一权责原则（SRP）[5] 认为，方法/类/组件应当只有一个修改的理由。并发设计自身足够复杂到成为修改的理由，所以也该从其他代码中分离出来。不幸的是，并发实现细节常常直接嵌入到其他生产代码中。下面是要考虑的一些问题：

- 并发相关代码有自己的开发、修改和调优生命周期；
- 开发相关代码有自己要对付的挑战，和非并发相关代码不同，而且往往更为困难；
- 即便没有周边应用程序增加的负担，写得不好的并发代码可能的出错方式数量也已经足具挑战性。

建议：分离并发相关代码与其他代码 [6] 。

13.3.2　推论：限制数据作用域

如我们所见，两个线程修改共享对象的同一字段时，可能互相干扰，导致未预期的行为。解决方案之一是采用synchronized关键字在代码中保护一块使用共享对象的临界区（critical section）。限制临界区的数量很重要。更新共享数据的地方越多，就越可能：

- 你会忘记保护一个或多个临界区——破坏了修改共享数据的代码；
- 得多花力气保证一切都受到有效防护（破坏了DRY原则[7]）；
- 很难找到错误源，也很难判断错误源。

建议：谨记数据封装；严格限制对可能被共享的数据的访问。

13.3.3 推论：使用数据复本

避免共享数据的好方法之一就是一开始就避免共享数据。在某些情形下，有可能复制对象并以只读方式对待。在另外的情况下，有可能复制对象，从多个线程收集所有复本的结果，并在单个线程中合并这些结果。

如果有避免共享数据的简易手段，结果代码就会大大减少导致错误的可能。你可能会关心创建额

外对象的成本。值得试验一下看看那是否真是个问题。然而，假使使用对象复本能避免代码同步执行，则因避免了锁定而省下的价值有可能补偿得上额外的创建成本和垃圾收集开销。

13.3.4 推论：线程应尽可能地独立

让每个线程在自己的世界中存在，不与其他线程共享数据。每个线程处理一个客户端请求，从不共享的源头接纳所有请求数据，存储为本地变量。这样一来，每个线程都像是世界中的唯一线程，没有同步需要。

例如，HttpServlet的子类接收所有以参数形式传递给doGet和doPost方法的信息。每个Servlet都像拥有独立虚拟机一般运行。只要Servlet中的代码只使用本地变量，Servlet就不会导致同步问题。当然，多数使用Servlet的应用程序最终都还是会用到类似数据库连接之类的共享资源。

建议：尝试将数据分解到可被独立线程（可能在不同处理器上）操作的独立子集。

13.4　了解Java库

相对于之前的版本，Java 5提供了许多并发开发方面的改进。在用Java 5编写线程代码时，要注意以下几点：

- 使用类库提供的线程安全群集；
- 使用executor框架（executor framework）执行无关任务；
- 尽可能使用非锁定解决方案；
- 有几个类并不是线程安全的。

线程安全群集

当Java还年轻时， Doug Lea编写了*Concurrent Programming in Java*（中译版《Java并发编程》）教程[8]，同时开发了几个线程安全群集，这些代码后来成为JDK中java.util.concurrent包的一部分。该代码包中的群集对于多线程解决方案是安全的，执行良好。实际上，在几乎所有情况下，ConcurrentHashMap实现都比HashMap表现得好。它还支持同步并发读写，也拥有支持非线程安全的合成操作的方法。如果部署环境是Java 5，可以采用ConcurrentHashMap。

还有几个支持高级并发设计的类。以下是其中一小部分，如表13-1所示。

表**13-1**　支持高级并发设计的类（部分）

ReentrantLock	可在一个方法中获取、在另一方法中释放的锁
Semaphore	经典的“信号”的一种实现，有计数器的锁
CountDownLatch	在释放所有等待的线程之前，等待指定数量事件发生的锁。这样，所有线程都平等地几乎同时启动

建议：检读可用的类。对于Java，掌握java.util.concurrent、java.util.concurrent.atomic和java.util.concurrent.locks。

13.5 了解执行模型

有几种在并发应用中切分行为的途径。要讨论这些途径，我们需要理解一些基础定义，如表13-2所示。

表13-2 基础定义

限定资源	并发环境中有着固定尺寸或数量的资源。例如数据库连接和固定尺寸读/写缓存等
互斥	每一时刻仅有一个线程能访问共享数据或共享资源
线程饥饿	一个或一组线程在很长时间内或永久被禁止。例如，总是让执行得快的线程先运行，假如执行得快的线程没完没了，则执行时间长的线程就会“挨饿”
死锁	两个或多个线程互相等待执行结束。每个线程都拥有其他线程需要的资源，得不到其他线程拥有的资源，就无法终止
活锁	执行次序一致的线程，每个都想要起步，但发现其他线程已经“在路上”。由于竞步的原因，线程会持续尝试起步，但在很长时间内却无法如愿，甚至永远无法启动

有了这些定义，我们就能讨论在并发编程中用到的几种执行模型了。

13.5.1 生产者-消费者模型[9]

一个或多个生产者线程创建某些工作，并置于缓存或队列中。一个或多个消费者线程从队列中获

取并完成这些工作。生产者和消费者之间的队列是一种限定资源 。

13.5.2 读者-作者模型 [10]

当存在一个主要为读者线程提供信息源，但只偶尔被作者线程更新的共享资源，吞吐量就会是个问题。增加吞吐量，会导致线程饥饿和过时信息的累积。更新会影响吞吐量。协调读者线程，不去读作者线程正在更新的信息（反之亦然），这是一种辛苦的平衡工作。作者线程倾向于长期锁定许多读者线程，从而导致吞吐量问题。

挑战之处在于平衡读者线程和作者线程的需求，实现正确操作，提供合理的吞吐量，避免线程饥饿。

13.5.3 宴席哲学家 [11]

想象一下，一群哲学家环坐在圆桌旁。每个哲学家的左手边放了一把叉子。桌面中央摆着一大碗意大利面。哲学家们思索良久，直至肚子饿了。每个人都要拿起叉子吃饭。但除非手上有两把叉子，否则就没法进食。如果左边或右边的哲学家已经取用一把叉子，中间这位就得等到别人吃完、放回叉

子。每位哲学家吃完后，就将两把叉子放回桌面，直到肚子再饿。

用线程代替哲学家，用资源代替叉子，就变成了许多企业级应用中进程竞争资源的情形。如果没有用心设计，这种竞争式系统就会遭遇死锁、活锁、吞吐量和效率降低等问题。

你可能遇到的并发问题，大多数都是这三个问题的变种。请研究并使用这些算法，这样，遇到并发问题时你就能有解决问题的准备了。

建议：学习这些基础算法，理解其解决方案。

13.6　警惕同步方法之间的依赖

同步方法之间的依赖会导致并发代码中的狡猾缺陷。Java语言有synchronized概念，可以用来保护单个方法。然而，如果在同一共享类中有多个同步方法，系统就可能写得不太正确了[12]。

建议：避免使用一个共享对象的多个方法。

有时必须使用一个共享对象的多个方法。在这种情况发生时，有3种写对代码的手段：

- 基于客户端的锁定——客户端代码在调用第一个方法前锁定服务端，确保锁的范围覆盖了调用最后一个方法的代码；
- 基于服务端的锁定——在服务端内创建锁定服务端的方法，调用所有方法，然后解锁。让客户端代码调用新方法；
- 适配服务端——创建执行锁定的中间层。这是一种基于服务端的锁定的例子，但不修改原始服务端代码。

13.7　保持同步区域微小

关键字synchronized制造了锁。同一个锁维护的所有代码区域在任一时刻保证只有一个线程执行。锁是昂贵的，因为它们带来了延迟和额外开销。所以我们不愿将代码扔给synchronized语句了事。另一方面，临界区 [13] 应该被保护起来。所以，应该尽可能少地设计临界区。

有些天真的程序员想通过扩大临界区面积达到这个目的。然而，将同步延展到最小临界区范围之外，会增加资源争用、降低执行效率 [14] 。

建议：尽可能减小同步区域。

13.8　很难编写正确的关闭代码

编写永远运行的系统，与编写运行一段时间后平静地关闭的系统是两码事。

平静关闭很难做到。常见问题与死锁[15]有关，线程一直等待永远不会到来的信号。

例如，想象一个系统中有个父线程分裂出数个子线程，父线程等待所有子线程结束，然后释放资源并关闭。如果其中一个子线程发生死锁会怎样？父线程将一直等待下去，而系统就永远不能关闭。

或者，考虑一个被指示 关闭的类似系统。父线程告知全体子线程放弃任务并结束。如果其中两个子线程正以生产者/消费者模型操作会怎样呢？假设生产者线程从父线程处接收到信号，并迅速关闭。消费者线程可能还在等待生产者线程发来消息，于是就被锁定在无法接收到关闭信号的状态中。它会死等生产者线程，永不结束，从而导致父线程也无法结束。

这类情形并非那么不常见。如果你要编写涉及平静关闭的并发代码，请多预留一些时间搞对关闭过程。

建议 ：尽早考虑关闭问题，尽早令其工作正常。这会花费比你预期更多的时间。检视既有算法，因为这可能会比想象中难得多。

13.9　测试线程代码

证明代码的正确性不切实际。测试并不能确保正确性。然而，好的测试却能尽量降低风险。这对于所有单线程解决方案都是对的。当有两个或多个线程使用同一代码段和共享数据，事情就变得非常复杂了。

建议 ：编写有潜力曝露问题的测试，在不同的编程配置、系统配置和负载条件下频繁运行。如果测试失败，跟踪错误。别因为后来测试通过了后来的运行就忽略失败。

有一大堆问题要考虑。下面是一些精练的建议：

- 将伪失败看作可能的线程问题；
- 先使非线程代码可工作；
- 编写可插拔的线程代码；
- 编写可调整的线程代码；
- 运行多于处理器数量的线程；
- 在不同平台上运行；
- 调整代码并强迫错误发生。

13.9.1 将伪失败看作可能的线程问题

线程代码导致“不可能失败的”失败。多数开发者缺乏有关线程如何与其他代码（可能由其他作者编写）互动的直觉。线程代码中的缺陷可能在一千或一百万次执行中才会显现一次。重复执行想要复现问题令人沮丧。所以开发者常常会将失败归咎于宇宙射线、硬件错误或其他“偶发事件”。最好假设这种偶发事件根本不存在。“偶发事件”被忽略得越久，代码就越有可能搭建于不完善的基础之上。

建议：不要将系统错误归咎于偶发事件。

13.9.2 先使非线程代码可工作

这看起来太浅显，但强调一下不无益处。确保线程之外的代码可工作。通常，这意味着创建由线程调用的POJO。POJO与线程无涉，所以可在线程环境之外测试。能放进POJO中的代码越多越好。

建议：不要同时追踪非线程缺陷和线程缺陷。确保代码在线程之外可工作。

13.9.3 编写可插拔的线程代码

编写可在数个配置环境下运行的线程代码：

- 单线程与多个线程在执行时不同的情况；
- 线程代码与实物或测试替身互动；
- 用运行快速、缓慢和有变动的测试替身执行；
- 将测试配置为能运行一定数量的迭代。

建议：编写可插拔的线程代码，这样就能在不同的配置环境下运行。

13.9.4　编写可调整的线程代码

要获得良好的线程平衡，常常需要试错。一开始，在不同的配置环境下监测系统性能。要允许线程数量可调整。在系统运行时允许线程发生变动。允许线程依据吞吐量和系统使用率自我调整。

13.9.5　运行多于处理器数量的线程

系统在切换任务时会发生一些事。为了促使任务交换的发生，运行多于处理器或处理器核心数量的线程。任务交换越频繁，越有可能找到错过临界区或导致死锁的代码。

13.9.6　在不同平台上运行

2007年，我们做了一套关于并发编程的课程。该课程主要在OS X下开发，在运行于虚拟机的

Windows XP上展示。用于演示的测试失败条件，在OS X上要比在XP上失败得更频繁。

被测试的代码已知是不正确的。这正强调了不同操作系统有着不同线程策略的事实，不同的线程策略影响了代码的执行。在不同环境中，多线程代码的行为也不一样[16]。应该在所有可能部署的环境中运行测试。

建议：尽早并经常地在所有目标平台上运行线程代码。

13.9.7 装置试错代码

并发代码中藏有缺陷，这并不罕见。简单的测试往往无法曝露这些缺陷。实际上，缺陷经常隐藏于一般处理过程中。可能好几个小时、好几天甚至好几个星期才会跳出来一次!

线程中的缺陷之所以如此不频繁、偶发、难以重现，是因为在几千个穿过脆弱区域的可能路径当中，只有少数路径会真的导致失败。经过会导致失败的路径的可能性惊人地低。所以，侦测与调试也非常之难。

怎么才能增加捕捉住如此罕见之物的机会？可

以装置代码，增加对Object.wait( )、Object.sleep( )、Object.yield( )和Object.priority( )等方法的调用，改变代码执行顺序。

这些方法都会影响执行顺序，从而增加了侦测到缺陷的可能性。有问题的代码，最好尽早、尽可能多地通不过测试。

有两种装置代码的方法：

- 硬编码；
- 自动化。

13.9.8　硬编码

你可以手工向代码中插入wait( )、sleep( )、yield( )和priority( )的调用。在测试某段棘手的代码时，正当如此操作。

下面是个例子：

```
public synchronized String nextUrlOrNull() {
  if(hasNext()) {
    String url = urlGenerator.next();
    Thread.yield(); // inserted for testing.
    updateHasNext();
    return url;
  }
  return null;
}
```

插入对yield( )的调用，将改变代码的执行路径，由此而可能导致代码在以前未失败过的地方失败。如果代码的确出错，那并非是因为你插入了yield( )方法调用[17]。代码出错了，这便是失败的原因。

这种手法有许多毛病：

- 你得手工找到合适的地方来插入方法调用；
- 你怎么知道在哪里插入调用、插入什么调用？
- 不必要地在产品环境中留下这类代码，将拖慢代码执行速度；
- 这是种无的放矢的手段。你可能找不到缺陷。实际上，这不在你把握之中。

我们所需要的，是一种在测试中但不在生产中实现的手段。我们还需要为多次运行轻易地调整配置，从而增加总的发现错误机会。

无疑，如果将系统分解为对线程及控制线程的类一无所知的POJO，就能更容易地找到装置代码的位置。而且，还能创建许多个以不同方式调用sleep、yield等方法的POJO测试。

13.9.9 自动化

可以使用Aspect-Oriented Framework、CGLIB或ASM之类工具通过编程来装置代码。例如，可以使用有单个方法的类：

```
public class ThreadJigglePoint {
  public static void jiggle() {
  }
}
```

可以在代码的不同位置调用这个方法：

```
public synchronized String nextUrlOrNull() {
  if(hasNext()) {
       ThreadJiglePoint.jiggle();
       String url = urlGenerator.next();
       ThreadJiglePoint.jiggle();
       updateHasNext();
       ThreadJiglePoint.jiggle();
       return url;
  }
  return null;
}
```

如此，你就得到了一个随机选择无所作为、睡眠或让步的方面。

或者，想象ThreadJigglePoint类有两种实现。第一种实现jiggle什么都不做，在生产环境中使用。

第二种实现生成一个随机数，在睡眠、让步或径直执行间做选择。如果上千次地做这种随机测试，大概就能找到一些缺陷的根源。假如测试都通过了，至少你可以说自己已谨慎对待。这种方法看似有点过于简单，但确是替代复杂工具的一种可选方案。

有一种叫做ConTest [18] 的工具，由IBM开发，能做类似的事情，但做法却稍微复杂些。

要点是让代码“异动”，从而使线程以不同次序执行。编写良好的测试与“异动”相组合，能有效地增加发现错误的机会。

建议：使用异动策略搜出错误。

13.10　小结

并发代码很难写正确。加入多线程和共享数据后，简单的代码也会变成噩梦。要编写并发代码，就得严格地编写整洁的代码，否则将面临微细和不频繁发生的失败。

第一要诀是遵循单一权责原则。将系统切分为分离了线程相关代码和线程无关代码的POJO。确保在测试线程相关代码时只是在测试，没有做其他事情。线程相关代码应该保持短小和目的集中。

了解并发问题的可能原因：对共享数据的多线程操作，或使用了公共资源池。类似平静关闭或停止循环之类边界情况尤其棘手。

学习类库，了解基本算法。理解类库提供的与基础算法类似的解决问题的特性。

学习如何找到必须锁定的代码区域并锁定之。不要锁定不必锁定的代码。避免从锁定区域中调用其他锁定区域。这需要深刻理解某物是否已共享。尽可能减少共享对象和共享范围。修改对象的设计，向客户代码提供共享数据，而不是迫使客户代码管理共享状态。

问题会跳出来。那种在早期没跳出来的问题往往是偶发的。这种所谓偶发问题，通常仅在高负载下出现或者偶然出现。所以，你要能在不同平台上、以不同配置持续重复运行线程代码。跟随TDD三要则而来的可测试性意味着某种程度的可插拔性，从而提供了在大量不同配置下运行代码的必要支持。

如果花点时间装置代码，就能极大地提升发现错误代码的机会。可以手工做，也可以使用某种自动化技术。尽早这么做。在将线程代码投入生产环境前，就要尽可能多地运行它。

只要采用了整洁的做法，做对的可能性就有翻天覆地的提高。

13.11 文献

[Lea99]：*Concurrent Programming in Java: Design Principles and Patterns* , 2d. ed., Doug Lea, Prentice Hall, 1999.

[PPP]：*Agile Software Development: Principles, Patterns, and Practices* , Robert C. Martin, Prentice Hall, 2002.

[PRAG]：*The Pragmatic Programmer* , Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

---

[1] 原注：来自私人邮件。

[2] 原注：宇宙射线、狼来了等。（译者按：作者在这里开了个小玩笑。程序员常把不能复现的程序错误的原因归结为宇宙射线等偶发性和无法修正的问题。）

[3] 原注：见后文“深入挖掘”一节。

[4] 原注：见后文“路径数量”一节。

[5]　原注：[PPP]。

[6]　原注：参见后文“客户端/服务器的例子”一节。

[7]　原注：[PRAG]。

[8]　原注：[Lea99]。

[9]　原注：http://en.wikipedia.org/wiki/Producer-consumer 。

[10]　原注：http://en.wikipedia.org/wiki/Readers-writers_problem 。

[11]　原注：http://en.wikipedia.org/wiki/Dining_philosophers_ 。

[12]　原注：参见后文“方法之间的依赖可能破坏同步代码”一节。

[13]　原注：临界区是为了确保程序正确而要阻止同时使用的代码区域。

[14]　原注：见后文“增加吞吐量”一节。

[15]　原注：参见附录A“死锁”一节。

[16]　原注：你是否知道，Java的线程模型并不保证线程抢先？现代操作系统支持抢先线程，所以你可以“免费”获得这一特性。即便如此，JVM也没有做出保证。

[17]　原注：严格说来并非如此。JVM不保证抢先线程，故在不抢占线程的系统上，某个特殊的算法可能一直能工作。反之亦然，但会有其他的原因影响。

[18]　原注：http://www.alphaworks.ibm.com/tech/contest 。

第14章　逐步改进

对一个命令行参数解析程序的案例研究



本章研究一个逐步改进的案例。你将看到一个开始还不错，规模扩大后即出问题的模块。你还将看到这个模块是如何被重构得整洁起来的。

我们中的大多数人都会遇到解析命令行参数的情况。如果没有就手的工具，就得遍历传入main函数的字符串数组。有一些不同来源的好工具，但没有一个是最符合要求的。所以，我当然要自己写一个。我把它叫做Args。

Args非常易于使用。你只要简单地用输入参数和格式化字符串构造Args类，再向Args实体询问参数值即可。看看下面的简单例子：

代码清单14-1　Args的简单用法

```
public static void main(String[] args) {
  try {
    Argsarg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    intport = arg.getInt('p');
    Stringdirectory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argumenterror:%s\n", e.errorMessage());
  }
}
```

可以看到这有多简单。我们只是用两个参数创建了Args类的一个实体。第一个参数是格式字符串，或范式 字符串：l,p#,d*。它定义了三个命令行参数。第一个，-l，是一个布尔值参数。第二个，-p，是一个整数参数。第三个，-d，是一个字符串参数。向Args构造器传入的第二个参数就是向main传入的命令行参数数组。

如果构造器正常返回，没有抛出ArgsException异常，则命令行参数已传入，Args实体随时待命。使用getBoolean、getInteger和getString等方法，可以用参数名称获得参数值。

不管是格式化字符串或命令行参数出现问题，就会抛出一个ArgsException异常。可以从该异常的errorMessage中获得关于错误的描述。

14.1 Args的实现

代码清单14-2是Args类的实现。请仔细阅读。我在代码风格和结构上花了大力气，使之值得仿效。

代码清单14-2 Args.java

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.Err
orCode.*;
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;

  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException
 {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();

    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
  }

  private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
      if (element.length() > 0)
        parseSchemaElement(element.trim());
  }

  private void parseSchemaElement(String element) throws ArgsExc
eption {
```

```
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
      marshalers.put(elementId, new StringArrayArgumentMarshaler
());
    else
      throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId
, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws Ar
gsException {
    if (!Character.isLetter(elementId))
      throw new ArgsException(INVALID_ARGUMENT_NAME, elementId,
null);
  }

  private void parseArgumentStrings(List<String> argsList) throw
s ArgsException
  {
    for (currentArgument = argsList.listIterator(); currentArgum
ent.hasNext();)
  {
      String argString = currentArgument.next();
      if (argString.startsWith("-")) {
        parseArgumentCharacters(argString.substring(1));
      } else {
        currentArgument.previous();
        break;
      }
    }
  }

  private void parseArgumentCharacters(String argChars) throws A
rgsException {
    for (int i = 0; i < argChars.length(); i++)
```

```
    parseArgumentCharacter(argChars.charAt(i));
  }

  private void parseArgumentCharacter(char argChar) throws ArgsE
xception {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
      throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null
);
    } else {
      argsFound.add(argChar);
      try {
        m.set(currentArgument);
      } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
      }
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public int nextArgument() {
    return currentArgument.nextIndex();
  }

  public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg)
);
  }

  public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg))
;
  }

  public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg)
);
  }

  public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg))
```

```
;
  }

  public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(
arg));
  }
}
```

注意，你可以从上到下阅读这些代码，不用跳来跳去，也不用先看后面的部分。唯一需要先看的是ArgumentMarshaler的定义，这部分我有意省略了。仔细看这段代码，你应该能理解ArgumentMarshaler接口是什么，其派生类做什么。下面我将向你展示一部分（如代码清单14-3～14-6所示）。

代码清单14-3　ArgumentMarshaler.java

```
public interface ArgumentMarshaler {
   void set(Iterator<String> currentArgument) throws ArgsExce
ption;
}
```

代码清单14-4　BooleanArgumentMarshaler.java

```
public class BooleanArgumentMarshaler implements ArgumentMarsha
ler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsE
xception {
```

```
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

代码清单14-5　StringArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.Err
orCode.*;

public class StringArgumentMarshaler implements ArgumentMarshal
er {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsE
xception {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

代码清单14-6　IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.Err
orCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarsha
ler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsE
xception {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

ArgumentMarshaler的其他派生类以同样的模式处理double和String数组，一一列出反而阻碍行文。你可以练习自己实现它们。

还有些信息可能会困扰你：错误码常量的定义。这些是在ArgsException类（代码清单14-7）中定义的。

代码清单14-7　ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.Err
orCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}
  public ArgsException(String message) {  super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParamete
r) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode,
                           char errorArgumentId,String errorParamete
r) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
```

```
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() {
    switch (errorCode) {
      case OK:
        return "TILT:  Should  not  get  here.";
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument  -%c  unexpected.", errorA
rgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for
 -%c.",
                              errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument  -%c  expects  an  integer
  but  was  '%s'.",
                                errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter fo
r -%c.",
                                errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but w
as '%s'.",
                                errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for
 -%c.",
                                errorArgumentId);
      case INVALID_ARGUMENT_NAME:
        return String.format("'%c' is not  a valid argument name.
",
                                errorArgumentId);

      case INVALID_ARGUMENT_FORMAT:
        return String.format("'%s' is not a valid argument format
```

```
.",
                                errorParameter);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_AR
GUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
  }
```

为了充实这么一个简单概念的细节，需要如此多代码，这很值得注意。原因之一是我们使用了Java这种唠叨型语言。作为一种静态类型语言，需要大量语句才能满足类型系统的要求。在Ruby、Python或Smalltalk等语言中，程序会短很多 [1] 。

请再次阅读这段代码。特别留意命名方式、函数大小和代码格式。如果你是经验丰富的程序员，可能会对风格或结构有着这样或那样的不同观点。不过，希望你认为这段程序总体上编写良好，有着整洁的结构。

例如，如何增加新参数类型，如日期或复杂数字参数。其实现手段很清楚，而且只需要花一点点力气即可。简言之，只需要从ArgumentMarshaler派生一个新类，写一个新的getXXX函数，在

parseSchemaElement函数中添加一个新的case语句。可能还需要添加新的ArgsException.Errorcode和新错误信息。

我怎么做的？

先放松一下神经。这段程序并非从一开始就写成现在的样子。更重要的是，我也没指望你能够一次过写出整洁、漂亮的程序。如果说我们从过去几十年里面学到什么东西的话，那就是编程是一种技艺甚于科学的东西。要编写整洁代码，必须先写肮脏代码，然后再清理它 。

你应该不会对此感到惊讶。我们在小学就学过这条真理了。那时，老师（通常是徒劳地）努力让我们写作文草稿。他们告诉我们，我们应该先写草稿，再写二稿，一次又一次地草撰，直至写出终稿。他们尽力告诉我们，写出好作文是一个逐步改进的过程。

多数新手程序员（就像多数小学生一样）没有特别认真地遵循这个建议。他们相信，首要任务是写出能工作的程序。只要程序“能工作”，就转移到下一个任务上，而那个“能工作”的程序就留在了最后那个所谓“能工作”的状态。多数老手程序员都知道，这是一种自毁行为。

14.2　Args：草稿

代码清单14-8展示了Args类的一个早期版本。它“能工作”，但却很烂。

代码清单14-8　Args.java（初稿）

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Chara
cter>();
  private Map<Character, Boolean> booleanArgs =
  new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Charac
ter, String>();
  private Map<Character, Integer> intArgs = new HashMap<Characte
r, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPE
CTED_ARGUMENT

  public Args(String schema, String[] args) throws ParseExceptio
n {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }
```

```
  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {

      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseEx
ception {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
      parseIntegerSchemaElement(elementId);
    } else {
      throw new ParseException
        (String.format("Argument:  %c  has  invalid  format:  %s.
",
                       elementId,elementTail),0);
    }
  }

  private void validateSchemaElementId(char elementId) throws Pa
rseException {
    if (!Character.isLetter(elementId)) {
```

```
      throw new ParseException(
          "Bad character:"+elementId+"in Args format: "+schema,0)
;
    }
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
    for (currentArgument = 0; currentArgument < args.length; cur
rentArgument++) {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
```

```
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

  private boolean setArgument(char argChar) throws ArgsException
 {
    if (isBooleanArg(argChar))
      setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
      setStringArg(argChar);
    else if (isIntArg(argChar))
      setIntArg(argChar);
    else
      return false;

    return true;
  }

  private boolean isIntArg(char argChar) {return intArgs.contain
sKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;

      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
```

```
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}

private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}

private boolean isStringArg(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
```

```
      throw new Exception("TILT:  Should  not  get  here.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Could  not  find  string  parameter
for  -%c.",
                              errorArgumentId);

    case INVALID_INTEGER:
      return String.format("Argument  -%c  expects  an  integer
but  was  '%s'.",
                             errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could  not  find  integer  parameter
for  -%c.",
                              errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s)  -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append("  unexpected.");

  return message.toString();
}

private boolean falseIfNull(Boolean b) {
  return b != null && b;
}

private int zeroIfNull(Integer i) {
  return i == null ? 0 : i;
}

private String blankIfNull(String s) {
  return s == null ? "" : s;
}

public String getString(char arg) {
  return blankIfNull(stringArgs.get(arg));
}
```

```
  public int getInt(char arg) {
    return zeroIfNull(intArgs.get(arg));
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }
}
```

希望你看到这段乱七八糟的代码时，第一反应是“他没就此罢手，真令人高兴！”如果你这么想，不如想想其他人对你留置在草稿形态的代码的想法吧。

实际上，“草稿”大概会是你对这段代码的最高评价。它显然还需打磨。实体变量的数量多到吓人。诸如TILT之类奇怪的字符串，HashSet和TreeSets，还有那些try-catch-catch代码块，组成了一个烂摊子。

我不想写出一个烂摊子。我也一直想保持一切

有序。从函数和变量命名，以及程序的粗略架构中，你可以看出这一点。不过，显然我没能做到。

混乱是逐渐产生的。更早的版本并不如此肮脏。例如，代码清单14-9展示了一个早期版本代码，那时只支持Boolean参数。

代码清单14-9　Args.java（只支持Boolean）

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Chara
cter>();
  private Map<Character, Boolean> booleanArgs =
   new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
```

```
}

private boolean parseSchema() {
  for (String element : schema.split(",")) {
    parseSchemaElement(element);
  }

  return true;
}

private void parseSchemaElement(String element) {
  if (element.length() == 1) {
    parseBooleanSchemaElement(element);
  }
}

private void parseBooleanSchemaElement(String element) {
  char c = element.charAt(0);
  if (Character.isLetter(c)) {
    booleanArgs.put(c, false);
  }
}

private boolean parseArguments() {
  for (String arg : args)
    parseArgument(arg);
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (isBoolean(argChar)) {
    numberOfArguments++;
    setBooleanArg(argChar, true);
  } else
    unexpectedArguments.add(argChar);
```

```
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return numberOfArguments;
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s)  -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append("  unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

尽管你可能对这段代码很不满意，其实它并非如此之烂。它精练、简单，易于理解。然而，在这段代码中很容易找到后面烂摊子的根源。很清楚能看到小问题如何变成大混乱的。

注意，后来的混乱代码只比这个版本多支持两种参数类型：String和integer。只增加两种参数类型支持，就对代码产生了如此巨大的负面影响。它从某种可维护之物变成了满是缺陷的东西。

我逐步添加了对这两种参数类型的支持。首先，我添加对String参数的支持，就像这样：

代码清单14-10　Args.java（Boolean和String）

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Chara
cter>();
  private Map<Character, Boolean> booleanArgs =
     new HashMap<Character, Boolean>();

  private Map<Character, String> stringArgs =
     new HashMap<Character, String>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgument = '\0';
```

```
  enum ErrorCode {
    OK, MISSING_STRING}

  private ErrorCode errorCode = ErrorCode.OK;

  public Args(String schema, String[] args) throws ParseExceptio
n {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseEx
ception {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
  }

  private void validateSchemaElementId(char elementId) throws Pa
rseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
```

```
      "Bad  character:" + elementId + "in  Args  format:  " + sch
ema, 0);
    }

  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; cur
rentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (setArgument(argChar))
      argsFound.add(argChar);
```

```
  else {
    unexpectedArguments.add(argChar);
    valid = false;
  }
}

private boolean setArgument(char argChar) {
  boolean set = true;
  if (isBoolean(argChar))
    setBooleanArg(argChar, true);
  else if (isString(argChar))
    setStringArg(argChar, "");
  else
    set = false;

  return set;
}

private void setStringArg(char argChar, String s) {
  currentArgument++;
  try {

    stringArgs.put(argChar, args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgument = argChar;
    errorCode = ErrorCode.MISSING_STRING;
  }
}

private boolean isString(char argChar) {
  return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return argsFound.size();
}
```

```
  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      switch (errorCode) {
        case MISSING_STRING:
          return String.format("Could  not  find  string  paramete
r  for  -%c.",
                               errorArgument);
        case OK:
          throw new Exception("TILT:  Should  not  get  here.");
      }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s)  -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append("  unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }
```

```
  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
}
```

你可以看到，代码开始失去控制。还算不上可怕，但混乱已经开始生长。已经出现了一堆东西，不过还没烂掉。增加对整数参数类型的支持后，那堆东西就真的变质腐烂了。

14.2.1　所以我暂停了

还有至少两种参数类型要添加，而且情形一定会更加糟糕。如果一味蛮干，大概也能让它工作，不过就会留下一大堆要调整的混乱。如果希望代码结构一直可维护，现在就是调整的时机了。

所以我暂停添加特性，开始重构。由于刚添加了String和integer参数，我知道每种参数类型都需要在三个主要位置增加新代码。首先，每种参数类型都要有解析其范式元素、从而为该种类型选择HashMap的方法。其次，每种参数类型都需要在命

令行字符串中解析，然后再转换为真实类型。最后，每种参数类型都需要一个getXXX方法，按照其真实类型向调用者返回参数值。

许多种不同类型，类似的方法——听起来像是个类。ArgumentMarshaler的概念就是这样产生的。

14.2.2　渐进

毁坏程序的最好方法之一就是以改进之名大动其结构。有些程序永远不能从这种所谓“改进”中恢复过来。问题在于，很难让程序以“改进”之前的方式工作。

为了避免这种状况发生，我采用了测试驱动开发的规程。这种手法的核心原则之一是保持系统始终能运行。换言之，采用TDD，我不会允许做出破坏系统的修改。每次修改都必须保证系统能像以前一样工作。

我需要一套能随需运行、确保系统行为不会改动的自动化测试。在我搞出那个烂摊子的同时，也为Args类创建了一套单元测试和验收测试。单元测试用Java写成，采用JUnit管理。验收测试用FitNesse以wiki页形式写成。我可以随时运行这些测试，如果测试通过，就能打包票说系统以我期望的

方式工作。

于是我开始做出大量小规模修改。每次修改都将系统结构向ArgumentMarshaler概念的方向推动。而且每次修改后，系统都要能工作。第一个修改是在烂摊子末尾添加ArgumentMarshaler的轮廓。

代码清单14-11　向Args.java添加ArgumentMarshaler

```
private class ArgumentMarshaler {
  private boolean booleanValue = false;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMars
haler {
  }

  private class StringArgumentMarshaler extends ArgumentMarsh
aler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMars
haler {
  }
}
```

显然，这什么也不会破坏。于是我做了一点最简单的、破坏性尽可能小的修改。我修改了

HashMap，采用ArgumentMarshaler，使之支持Boolean参数。

```
private  Map<Character,  ArgumentMarshaler


>  booleanArgs  =
   new  HashMap<Character,  ArgumentMarshaler


>();
```

这个修改影响到少数语句，我很快就修正了。

```
...
private  void  parseBooleanSchemaElement(char  elementId)  {
booleanArgs.put(elementId,  new  BooleanArgumentMarshaler()


);
}
...
private  void  setBooleanArg(char  argChar,  boolean  value)  {
booleanArgs.get


(argChar).setBoolean


(value);
}
...
public  boolean  getBoolean(char  arg)  {
return  falseIfNull(booleanArgs.get(arg).getBoolean()


);
```

```
}
```

注意，这些修改正是在我之前提到的那些区域之内所做的：参数类型的parse、set和get操作。不幸的是，即便修改如此细微，有些测试还是会失败。仔细看getBoolean，可以看到如果用y去调用、而并没有y这个参数，则booleanArgs.get('y')就会返回null值，函数将抛出一个NullPointerException异常。函数falseIfNull用以防止这种状况发生，但我做出的修改却导致该函数无所作为。

渐进主义要求我在做其他修改之前迅速修正这个问题。修正并不费劲。我只是把对null值的检查移了个位置。再也不用检测bollean是否为null，而是检查ArgumentMarshaler是否为null。

首先，我移除了getBoolean函数中的falseIfNull调用。现在它没什么用了，所以我也删去了这个函数。测试还是以同样的方式失败，所以我确定没有引入新的错误。

```
public boolean getBoolean(char arg) {
  return booleanArgs.get(arg).getBoolean();
}
```

下一步，我把函数拆解为两行，并把ArgumentMarshaler放到它自己的名为argumentMarshaler的变量中[2]。我不在意变量名太长，但它却有点啰嗦，把函数搞得支离破碎。所以我把变量名缩短为am[N5]。

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am


  = booleanArgs.get(arg);
  return am


.getBoolean();
}
```

然后再放入检测null值的逻辑。

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null &&


  am.getBoolean();
}
```

14.3　字符串参数

添加String参数和添加boolean参数非常像。我要修改HashMap，让parse、set和get函数能工作。跟着就是按部就班，但我似乎该把所有的marshalling（编组）实现放到ArgumentMarshaler基类而不是派生类中。

```
  private Map<Character, ArgumentMarshaler


> stringArgs =
    new HashMap<Character, ArgumentMarshaler


>();
  ...
  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler()


);
  }
  ...
  private void setStringArg(char argChar) throws ArgsExcepti
on {
    currentArgument++;
    try {
      stringArgs.get


(argChar).setString


(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
```

```
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
  ...
  public String getString(char arg) {
    Args.ArgumentMarshaler am

  = stringArgs.get(arg);
    return am == null ?

  "" : am.getString();
  }
  ...
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {
    return booleanValue;
  }

 public void setString(String s) {

    stringValue = s;

 }
```

```
public String getString() {

  return stringValue == null ? "" : stringValue;

  }

}
```

同样，也是每次修改一个地方，持续运行测试。如果测试出错，在做下一个修改前确保通过。

现在你应该明白我的意图了。一旦我将当前的编组行为放到ArgumentMarshaler基类中，就会开始往派生类推入该行为。这样，在我逐渐修改程序的形状时，还能保持一切正常。

下一步显而易见，把int参数的相关功能放到ArgumentMarshaler里面。同样，也是照方抓药。

```
private Map<Character, ArgumentMarshaler>

 intArgs =
  new HashMap<Character, ArgumentMarshaler>
```

```
();
...
private void parseIntegerSchemaElement(char elementId) {
  intArgs.put(elementId, new IntegerArgumentMarshaler()


);
}
...
private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.get


(argChar).setInteger


(Integer.parseInt(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}
...
public int getInt(char arg) {
  Args.ArgumentMarshaler am =


  intArgs.get(arg);
  return am == null ? 0


: am.getInteger();
}
```

```
...
private class ArgumentMarshaler {
  private boolean booleanValue = false;
  private String stringValue;
  private int integerValue;



  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
    return stringValue == null ? "" : stringValue;
}

  public void setInteger(int i) {



     integerValue = i;



}




  public int getInteger() {



     return integerValue;
```

```
 }


}
```

当所有的编组操作都放到了ArgumentMarshaler中，我开始向派生类移植功能。第一步是把setBoolean函数放到BooleanArgumentMarshaler中，确保它能正确调用。所以我创建了一个抽象的set方法。

```
private abstract


 class ArgumentMarshaler {
protected


boolean booleanValue = false;
 private String stringValue;
 private int integerValue;

 public void setBoolean(boolean value) {
  booleanValue = value;
 }

 public boolean getBoolean() {
  return booleanValue;
 }

 public void setString(String s) {
  stringValue = s;
 }

 public String getString() {
```

```
  return stringValue == null ? "" : stringValue;
 }

 public void setInteger(int i) {
  integerValue = i;
 }

 public int getInteger() {
  return integerValue;
 }

 public abstract void set(String s);



 }
```

然后在BooleanArgumentMarshaler中实现set方法。

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
 public void set(String s) {



   booleanValue = true;



 }



}
```

最后，通过调用set，替换对setBoolean的调用。

```
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).set("true");



}
```

测试仍然全部通过。因为这次修改导致set函数放到了BooleanArgumentMarshaler里面，我就从ArgumentMarshaler基类删除了setBoolean方法。

注意，抽象函数set有一个String参数，但其在BooleanArgumentMarshaler中的实现却没有使用这个参数。之所以在这里放个参数，是因为我知道StringArgumentMarshaler和IntegerArgumentMarshaler可能会使用它。

跟着，我打算把get方法放到BooleanArgumentMarshaler中。这有点难看，因为返回类型必须是Object，且在这里需要转换为Boolean值。

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean)


am.get


();
}
```

为了编译通过，我把get函数加到ArgumentMarshaler中。

```
private abstract class ArgumentMarshaler {
...

  public Object get() {



    return null;



  }



}
```

这样一来，虽然可以编译，但却无法通过测试。只要将get修改为抽象方法，并在

BooleanArgumentMarshaler中实现，就能重新通过测试。

```
 private abstract class ArgumentMarshaler {
   protected boolean booleanValue = false;
   ...

   public abstract


  Object get();
 }

 Private class BooleanArgumentMarshaler extends ArgumentMarsh
aler {
   public void set(String s) {
     booleanValue = true;
 }

 public Object get() {



    return booleanValue;



 }



}
```

测试又通过了。get和set方法都已部署到BooleanArgumentMarshaler中！这样我就可以从ArgumentMarshaler里面移除旧的getBoolean函数，

把受保护的booleanValue变量向下移动到BooleanArgumentMarshaler，并将其设置为private。

对于String也照此办理。我修改了set和get的部署方式，删除无用的函数，并移动了变量。

```
 private void setStringArg(char argChar) throws ArgsExceptio
n {
   currentArgument++;
   try {
    stringArgs.get(argChar).set


(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
   valid = false;
   errorArgumentId = argChar;
   errorCode = ErrorCode.MISSING_STRING;
   throw new ArgsException();
 }
}
 …
 public String getString(char arg) {
   Args.ArgumentMarshaler am = stringArgs.get(arg);
   return am == null ? "" : (String)


  am.get


();
 }
 ...
 private abstract class ArgumentMarshaler {
   private int integerValue;

   public void setInteger(int i) {
    integerValue = i;
   }

   public int getInteger() {
```

```
    return integerValue;
  }

  public abstract void set(String s);

  public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarsh
aler {
  private boolean booleanValue = false;




  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarsha
ler {
  private String stringValue = "";




  public void set(String s) {
    stringValue = s;



  }

  public Object get() {
    return stringValue;



  }
}
```

```
 private class IntegerArgumentMarshaler extends ArgumentMarsh
aler {

   public void set(String s) {

   }

   public Object get() {
    return null;
   }
 }
}
```

最后，我为integer类型参数重复这个过程。这稍稍复杂一点，因为integer需要解析，而parse操作会抛出异常。不过结果会更好，因为NumberFormatException的概念在IntegerArgumentMarshaler中隐藏了。

```
private boolean isIntArg(char argChar) {return intArgs.conta
insKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.get(argChar).set


(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
```

```
  } catch (ArgsException


  e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e


;
  }
}
...
private void setBooleanArg(char argChar) {
  try {



     booleanArgs.get(argChar).set("true");
  } catch (ArgsException e) {



  }



}
...
public int getInt(char arg) {
  Args.ArgumentMarshaler am = intArgs.get(arg);
  return am == null ? 0 : (Integer)


  am.get


();
}
...
private abstract class ArgumentMarshaler {
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
```

```
}
...
private class IntegerArgumentMarshaler extends ArgumentMarsha
ler {
  private int intValue = 0;

  public void set(String s) throws ArgsException {

  try {

    intValue = Integer.parseInt(s);

   } catch (NumberFormatException e) {

   throw new ArgsException();

   }

}
  public Object get() {
   return intValue;

  }
}
```

测试当然继续通过。下一步，我要删掉算法顶端的三种不同Map。这样，整个系统就变得更通用了。不过，只是删除它们却无法达到目的，因为那样会破坏系统。反之，我为ArgumentMarshaler添加一个新的Map，然后再逐个修改那些方法，让方法调用这个新Map。

```
public class Args {
  ...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character,ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =



     new HashMap<Character, ArgumentMarshaler>()


;
...
  private void parseBooleanSchemaElement(char elementId) {
   ArgumentMarshalerm = new BooleanArgumentMarshaler();



   booleanArgs.put(elementId, m);
   marshalers.put(elementId, m);



  }

  private void parseIntegerSchemaElement(char elementId) {
   ArgumentMarshaler m = new IntegerArgumentMarshaler();
```

```
  intArgs.put(elementId, m);
  marshalers.put(elementId, m);
 }

 private void parseStringSchemaElement(char elementId) {
  ArgumentMarshaler m = new StringArgumentMarshaler()
;
  stringArgs.put(elementId,m);
  marshalers.put(elementId, m);
}
```

当然，测试还是通过了。接着，我把isBooleanArg：

```
private boolean isBooleanArg(char argChar) {
 return booleanArgs.containsKey(argChar);
}
```

修改成这样：

```
private boolean isBooleanArg(char argChar) {
 ArgumentMarshaler m = marshalers.get(argChar);
```

```
  return m instanceof BooleanArgumentMarshaler;



}
```

测试仍然通过。于是我修改了一下isIntArg和isStringArg。

```
private boolean isIntArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);



 return m instanceof IntegerArgumentMarshaler;



}
```

```
private boolean isStringArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof StringArgumentMarshaler;
}
```

测试继续通过。我跟着消除了对marshaler.get的重复调用：

```
private boolean setArgument(char argChar) throws ArgsExcepti
```

```
on {
  ArgumentMarshaler m = marshalers.get(argChar);



  if (isBooleanArg(m


))
    setBooleanArg(argChar);
  else if (isStringArg(m


))
    setStringArg(argChar);
  else if (isIntArg(m


))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(ArgumentMarshaler m


) {
  return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(ArgumentMarshaler m


) {
  return m instanceof StringArgumentMarshaler;
}

private boolean isBooleanArg(ArgumentMarshaler m


) {
  return m instanceof BooleanArgumentMarshaler;
```

```
}
```

存在三个isxxxArg方法毫无道理。所以我做了内联修改：

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(argChar);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
```

下一步，我开始在set函数中使用marshaler映射，停止使用另外三个映射映射。从boolean开始：

```
private boolean setArgument(char argChar) throws ArgsException
```

```
{
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(m


);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
...
private void setBooleanArg(ArgumentMarshaler m


) {
  try {
    m


.set("true"); // was: booleanArgs.get(argChar).set("true")


;
  } catch (ArgsException e) {
  }
}
```

测试通过，于是我如法炮制String和Integer参数。这样我就能把有些丑陋的异常管理代码整合到setArgument函数中。

```
private  boolean  setArgument(char  argChar)  throws  ArgsExcepti
```

```
on {
 ArgumentMarshaler m = marshalers.get(argChar);
 try {

    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m

);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m

);
    else
      return false;
  } catch (ArgsException e) {

    valid = false;

    errorArgumentId = argChar;

    throw e;

  }

  return true;
}

private void setIntArg(ArgumentMarshaler m
```

```
) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m


.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

private void setStringArg(ArgumentMarshaler m


) throws ArgsException {
  currentArgument++;
  try {
    m


.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
```

离彻底删除那3个旧映射的时机越来越近了。首先，我需要修改getBoolean函数：

```java
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean) am.get();
}
```

修改成这样：

```java
public boolean getBoolean(char arg) {
Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;

try {

b =

am != null && (Boolean) am.get();
} catch (ClassCastException e) {

b = false;

}

return b;

}
```

最后这个修改可能令人吃惊。为什么我会突然决定对付ClassCastException？原因是我有一组单元测试，还有用FitNesse编写的一组验收测试。FitNesse测试确认，如果用非布尔值参数调用getBoolean，应该返回false。可单元测试的结果不是这样。而到此时为止，我一直只调用单元测试[3]。

这次修改把另一个对boolean映射的使用抽离了：

```
private void parseBooleanSchemaElement(char elementId) {
  ArgumentMarshaler m = new BooleanArgumentMarshaler();
  booleanArgs.put(elementId, m);
  marshalers.put(elementId, m);
}
```

如此我们就能删除boolean映射。

```
public class Args {
...
```

```
    private  Map<Character,  ArgumentMarshaler>  booleanArgs  =

new  HashMap<Character,  ArgumentMarshaler>();

    private  Map<Character,  ArgumentMarshaler>  stringArgs  =
      new  HashMap<Character,  ArgumentMarshaler>();
    private  Map<Character,  ArgumentMarshaler>  intArgs  =
      new  HashMap<Character,  ArgumentMarshaler>();
    private  Map<Character,  ArgumentMarshaler>  marshalers  =
      new  HashMap<Character,  ArgumentMarshaler>();
...
```

接下来，我用同样的手法处理String和Integer参数，对boolean参数做了一点清理工作。

```
private  void  parseBooleanSchemaElement(char  elementId)  {
  marshalers.put(elementId,  new  BooleanArgumentMarshaler()

);
}

private  void  parseIntegerSchemaElement(char  elementId)  {
  marshalers.put(elementId,  new  IntegerArgumentMarshaler()

);
```

```
}

private void parseStringSchemaElement(char elementId) {
  marshalers.put(elementId, new StringArgumentMarshaler()


);
}
...
public String getString(char arg) {
Args.ArgumentMarshaler am = marshalers.


get(arg);
try {



return am == null ? "" : (String) am.get();
} catch (ClassCastException e) {



return "";



}



}

public int getInt(char arg) {
  Args.ArgumentMarshaler am = marshalers.


get(arg);
  try {



    return am == null ? 0 : (Integer) am.get();
 } catch (Exception e) {
```

```
   return 0;
 }
}
...
public class Args {
  ...
private Map<Character, ArgumentMarshaler> st ringArgs =
     new HashMap<Character, ArgumentMarshaler>();
private Map<Character, ArgumentMarshaler> intArgs =
     new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
```

```
    new HashMap<Character, ArgumentMarshaler>();
  ...
```

接着，由于那些parse方法没有太多事可做，我对它们进行了内联修改：

```
private void parseSchemaElement(String element) throws Parse
Exception {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    marshalers.put(elementId, new BooleanArgumentMarshaler());



  else if (isStringSchemaElement(elementTail))
    marshalers.put(elementId, new StringArgumentMarshaler());



  else if (isIntegerSchemaElement(elementTail)) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());



  } else {
    throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, e
lementTail), 0);
  }
}
```

行了，下面来看看全景吧。代码清单14-12展

示了Args类的现状。

代码清单14-12　Args.java（首次重构后）

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<C
haracter>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>(
);
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNE
XPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseExce
ption {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
```

```
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws Par
seException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler())
;
     else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler()
);

      else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler(
));
      } else {
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId
, elementTail), 0);
      }
  }

  private void validateSchemaElementId(char elementId) throws
  ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
          "Bad character:" + elementId + "in Args format: "
  + schema, 0);
    }
  }

  private boolean isStringSchemaElement(String elementTail) {
```

```
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail)
{
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail)
{
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; curre
ntArgument++) {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsExceptio
n {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsExceptio
n {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsExcepti
on {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
  }
```

```
  private boolean setArgument(char argChar) throws ArgsExcep
tion {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsExc
eption {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws Args
Exception {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
```

```
      throw new ArgsException();
    }
  }

  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true");
    } catch (ArgsException e) {
    }
  }

  public int cardinality() {
    return argsFound.size();
  }


  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter
  for -%c.",
                                errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer
  but was '%s'.",
                                errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer paramete
r for -%c.",
                                 errorArgumentId);
    }
    return "";
  }
```

```
private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" unexpected.");

  return message.toString();
}

public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}

public String getString(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
    return "";
  }
}


public int getInt(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? 0 : (Integer) am.get();
  } catch (Exception e) {
    return 0;
  }
}

public boolean has(char arg) {
  return argsFound.contains(arg);
}

public boolean isValid() {
```

```
    return valid;
  }

  private class ArgsException extends Exception {
  }

  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMars
haler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarsh
aler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMars
haler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
```

```
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
 }
}
```

功夫费尽，还是有点失望。程序结构好了一点，但在代码顶端还是有那一堆变量；在setArgument里面还是有那么恐怖的类型转换操作；而且那些set函数真的很丑陋。就别提那些错误处理操作了。前头要做的事还很多。

我真是想删掉setArgument里面那些类型转换操作[G23]。我想要setArgument只简单地调用ArgumentMarshaler.set。这意味着我需要将setIntArg、setStringArg和setBooleanArg推到合适的ArgumentMarshaler派生类里面。不过这有个问题。

仔细看setIntArg，你会发现，它使用了两个实体变量：args和currentArg。为了把setIntArg移到BooleanARgumentMarshaler里面，我得把这两个变量都作为函数参数传递过去。那种做法太烂了[F1]。我只想传递一个参数。幸运的是，有个简单的解决方法。可以把args数组转换为一个list，并向set函数传递一个Iterator。这花了我10步功夫，每次都通过了测试。不过我只向你展示结果。你应该能看出每个小修改步骤。

```
public class Args {
  private String schema;
  ~~private String[] args;~~






  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
```

```
 private  Iterator<String>


 currentArgument;
 private  char  errorArgumentId  =  '\0';
 private  String  errorParameter  =  "TILT";
 private  ErrorCode  errorCode  =  ErrorCode.OK;
private  List<String>  argsList;




 private  enum  ErrorCode  {
   OK,  MISSING_STRING,  MISSING_INTEGER,  INVALID_INTEGER,  UNEXPECTED_ARGUMENT}

 public  Args(String  schema,  String[]  args)  throws  ParseException  {
   this.schema  =  schema;
   argsList  =  Arrays.asList(args);



   valid  =  parse();
 }


 private  boolean  parse()  throws  ParseException  {
   if  (schema.length()  ==  0  &&  argsList.size()


 ==  0)
     return  true;
     parseSchema();
     try  {
       parseArguments();
     }  catch  (ArgsException  e)  {
     }
     return  valid;
 }
 ---
 private  boolean  parseArguments()  throws  ArgsException  {
   for  (currentArgument  =  argsList.iterator();


currentArgument.hasNext();


)  {
     String  arg  =  currentArgument.next();
     parseArgument(arg);
}

     retun true;
   }
   ---
   private  void  setIntArg(ArgumentMarshaler  m)  throws  ArgsException  {
     String  parameter  =  null;
     try  {
       parameter  =  currentArgument.next


();
```

```
      m.set(parameter);
    } catch (NoSuchElementException

e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next

());
    } catch (NoSuchElementException

e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

是这些简单的修改让测试保持通过。现在我们可以开始把set函数移植到合适的派生类中了。第一步，我要在setArgument中做以下修改：

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
 if (m == null)

    return false;

  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
```

```
      return  false;
```

```
  }  catch  (ArgsException  e)  {
    valid  =  false;
    errorArgumentId  =  argChar;
    throw  e;
  }
  return  true;
}
```

这个修改很重要，因为我们想要彻底删除那条if-else链。所以，需要把错误条件抽离。

现在可以开始移动set函数了。setBooleanArg函数很小，就从它开始。目标是让setBooleanArg函数只与BooleanArgumentMarshaler有关。

```
private  boolean  setArgument(char  argChar)  throws  ArgsException  {
  ArgumentMarshaler  m  =  marshalers.get(argChar);
  if  (m  ==  null)
    return  false;
  try  {
    if  (m  instanceof  BooleanArgumentMarshaler)
      setBooleanArg(m,  currentArgument


);
    else  if  (m  instanceof  StringArgumentMarshaler)
      setStringArg(m);
    else  if  (m  instanceof  IntegerArgumentMarshaler)
      setIntArg(m);
  }  catch  (ArgsException  e)  {
    valid  =  false;
    errorArgumentId  =  argChar;
    throw  e;

  }
  return  true;
}
---
```

```
private  void  setBooleanArg(ArgumentMarshaler  m,
                             Iterator<String>  currentArgument)
                             throws  ArgsException  {
  try  {
    m.set("true");
catch  (ArgsException  e)  {
}
}
```

我们不是刚把那个异常处理放进去吗？放进拿出是重构过程中常见的事。小步幅和保持测试通过，意味着你会不断移动各种东西。重构有点像是解魔方。需要经过许多小步骤，才能达到较大目标。每一步都是下一步的基础。

为什么要在setBooleanArg根本不需要的情况下向其传递iterator呢？因为setIntArg和setStringArg需要！还因为我打算通过ArgumentMarshaler中的抽象方法部署这三个函数，需要将其传递给setBooleanArg。

现在setBooleanArg没用了。如果ArgumentMarshaler中有个set函数，我们可以直接调用它。是时候打造那个函数了！第一步，在ArgumentMarshaler中添加抽象方法。

```
private abstract class ArgumentMarshaler {
 public abstract void set(Iterator<String> currentArgument)

                throws ArgsException;

  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}
```

当然，这会影响到所有派生类。所以，要逐个实现新方法。

```
 private class BooleanArgumentMarshaler extends ArgumentMarshaler {
   private boolean booleanValue = false;

   public void set(Iterator<String> currentArgument) throws ArgsException {

      booleanValue = true;

 }

   public void set(String s) {

booleanValue = true;

 }

   public Object get() {
     return booleanValue;
   }
 }

 private class StringArgumentMarshaler extends ArgumentMarshaler {
   private String stringValue = "";

   public void set(Iterator<String> currentArgument) throws ArgsException {

 }
```

```
 public void set(String s) {
  stringValue = s;
 }

 public Object get() {
  return stringValue;
 }
}

 private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {



 }




  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
 }
 }
```

现在可以删除setBooleanArg了！

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set


(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
  } catch (ArgsException e) {
```

```
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}
```

测试全都通过，而且set函数也部署到BooleanArgumentMarshaler里面了！

现在就能对String和Integer参数的处理做同样的修改。

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
      return false;
  try {
    if (m instanceof BooleanArgumentMarshaler)
      m.set(currentArgument);
    else if (m instanceof StringArgumentMarshaler)
      m.set(currentArgument);


    else if (m instanceof IntegerArgumentMarshaler)
      m.set(currentArgument);


   } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}---
private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {


      stringValue = currentArgument.next();


    } catch (NoSuchElementException e) {
```

```
      errorCode = ErrorCode.MISSING_STRING;

      throw new ArgsException();

    }

  }

  public void set(String s) {
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;

    try {

      parameter = currentArgument.next();

      set(parameter);

    } catch (NoSuchElementException e) {

      errorCode = ErrorCode.MISSING_INTEGER;

      throw new ArgsException();

    } catch (ArgsException e) {
```

```
      errorParameter = parameter;

      errorCode = ErrorCode.INVALID_INTEGER;

      throw e;

    }

  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

最后一击

：可以移除类型转换了！看招！

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
}
```

现在可以删掉IntegerArgumentMarshaler中那些过时的函数，做一下清理了。

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0

  public void set(Iterator<String> currentArgument) throws ArgsException {
     String
  parameter = null;
     try {
    parameter = currentArgument.next();
    intValue = Integer.parseInt


(parameter);
} catch (NoSuchElementException e) {
errorCode = ErrorCode.MISSING_INTEGER;
throw new ArgsException();
  } catch (NumberFormatException


e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}

  public Object get() {
    return intValue;
  }
}
```

还可以把ArgumentMarshaler修改为接口。

```
private interface


  ArgumentMarshaler {
void set(Iterator<String> currentArgument) throws ArgsException;
Object get();
}
```

现在来看看往这个结构中添加新的参数类型有多容易。只需要做少量修改，而且修改是被隔离的。首先，增加一个新的测试用例，检测double参数是否正常工作。

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

然后清理范式解析代码，为double参数类型添加##监测。

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length()


  == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*")


)
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#")


)
    marshalers.put(elementId, new IntegerArgumentMarshaler());

 else if (elementTail.equals("##"))



     marshalers.put(elementId, new DoubleArgumentMarshaler());



   else
    throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

下一步，编写DoubleArgumentMarshaler类。

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {

  private double doubleValue = 0;

public void set(Iterator<String> currentArgument) throws ArgsException {

 String parameter = null;

   try {

  parameter = currentArgument.next();

  doubleValue = Double.parseDouble(parameter);

 } catch (NoSuchElementException e) {

  errorCode = ErrorCode.MISSING_DOUBLE;

  throw new ArgsException();

 } catch (NumberFormatException e) {

  errorParameter = parameter;

  errorCode = ErrorCode.INVALID_DOUBLE;
```

```
    throw new ArgsException();
  }
  }

  public Object get() {
    return doubleValue;
  }
}
```

然后就得添加一个新的ErrorCode：

```
private enum ErrorCode {
  OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
 MISSING_DOUBLE, INVALID_DOUBLE}
```

还需要一个getDouble函数：

```java
public double getDouble(char arg) {

  Args.ArgumentMarshaler am = marshalers.get(arg);

  try {

    return am == null ? 0 : (Double) am.get();

  } catch (Exception e) {

    return 0.0;

  }

 }
```

全部测试都通过了！完全无痛。再来确保全部错误处理代码正确工作。下一个测试用例用来检测在向##参数传递一个不可解析的字符串时是否会返回错误。

```java
public void testInvalidDouble() throws Exception {
  Args args = new Args("x##", new String[] {"-x","Forty two"});
  assertFalse(args.isValid());
  assertEquals(0, args.cardinality());
  assertFalse(args.has('x'));
  assertEquals(0, args.getInt('x'));
  assertEquals("Argument -x expects a double but was 'Forty two'.",
                args.errorMessage());

}
---
public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
```

```
    throw new Exception("TILT: Should not get here.");
  case UNEXPECTED_ARGUMENT:
    return unexpectedArgumentMessage();
  case MISSING_STRING:
    return String.format("Could not find string parameter for -%c.",
                         errorArgumentId);
  case INVALID_INTEGER:
    return String.format("Argument -%c expects an integer but was '%s'.",
                          errorArgumentId, errorParameter);
  case MISSING_INTEGER:
    return String.format("Could not find integer parameter for -%c.",
                          errorArgumentId);
  case INVALID_DOUBLE:
    return String.format("Argument -%c expects a double but was '%s'.",
                         errorArgumentId, errorParameter);
  case MISSING_DOUBLE:
    return String.format("Could not find double parameter for -%c.",
                         errorArgumentId);
 }
 return "";
}
```

测试通过。下一个测试确保我们正确检测到遗漏的double参数。

```
public void testMissingDouble() throws Exception {
   Args args = new Args("x##", new String[]{"-x"});
   assertFalse(args.isValid());
```

```
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
                 args.errorMessage());
  }
```

测试如期通过。我们只是为了保持一切完整而编写这个测试。

异常代码很丑陋，不该在Args类中存在。我们也抛出ParseException，但那并不真的属于我们自己。那就把所有异常都塞到ArgsException类中，并将其移到它自己的模块里面。

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
```

```
  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public enum ErrorCode {

    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,

    MISSING_DOUBLE, INVALID_DOUBLE}

 }

 ---

public class Args {
  ...
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ArgsException.

ErrorCode errorCode = ArgsException.

ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException

{
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ArgsException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
```

```
    try {
      parseArguments();
    } catch (ArgsException

e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ArgsException

{
    ...
  }

  private void parseSchemaElement(String element) throws ArgsException

{
    ...
    else
      throw new ArgsException

(
        String.format("Argument: %c has invalid format: %s.",
                      elementId,elementTail));
  }

  private void validateSchemaElementId(char elementId) throws ArgsException

{
    if (!Character.isLetter(elementId)) {
      throw new ArgsException

(
        "Bad character:" + elementId + "in Args format: " + schema);
    }
  }

  ...

  private void parseElement(char argChar) throws ArgsException

  {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ArgsException

.ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }
```

```
  ...

  private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ArgsException


.ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException


  {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
        throw new ArgsException


();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException


.ErrorCode.INVALID_INTEGER;
        throw new ArgsException


();
      }
    }

    public Object get() {
      return intValue;
    }
  }

  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;
```

```
    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ArgsException


.ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ArgsException


.ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
}
```

很好。现在，Args抛出的唯一一个异常是ArgsException。把ArgsException移到它自己的模块中，意味着我们能把大量杂七杂八的错误支持代码从Args模块转移到这个模块。

现在我们完全把异常和错误代码从Args模块中隔离出来了。（如代码清单14-13～16所示。）为达到这一目标，大概做了30次小修改，每次修改都保持测试通过。

代码清单14-13 ArgsTest.java

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args("", new String[0]);
    assertEquals(0, args.cardinality());
  }
```

```
  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args("", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args("", new String[]{"-x", "-y"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }

  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args("*", new String[]{});
      fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                   e.getErrorCode());
      assertEquals('*', e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
      new Args("f~", new String[]{});
      fail("Args constructor should have throws exception");
} catch (ArgsException e) {
assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
    assertEquals('f', e.getErrorArgumentId());
    }
  }

  public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean('x'));
  }

  public void testSimpleStringPresent() throws Exception {
    Args args = new Args("x*", new String[]{"-x", "param"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals("param", args.getString('x'));
  }

  public void testMissingStringArgument() throws Exception {
    try {
      new Args("x*", new String[]{"-x"});
```

```
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSpacesInFormat() throws Exception {
    Args args = new Args("x, y", new String[]{"-xy"});
    assertEquals(2, args.cardinality());
    assertTrue(args.has('x'));
    assertTrue(args.has('y'));
  }

  public void testSimpleIntPresent() throws Exception {
    Args args = new Args("x#", new String[]{"-x", "42"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42, args.getInt('x'));
  }

  public void testInvalidInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }
  }

  public void testMissingInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[]{"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
  }

  public void testInvalidDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }
  }

  public void testMissingDouble() throws Exception {
```

```
    try {
      new Args("x##", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }
}
```

代码清单14-14 ArgsExceptionTest.java

```
public class ArgsExceptionTest extends TestCase {
  public void testUnexpectedMessage() throws Exception {
    ArgsException e =
    new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                      'x', null);
    assertEquals("Argument -x unexpected.", e.errorMessage());
  }

  public void testMissingStringMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                        'x', null);
    assertEquals("Could not find string parameter for -x.", e.errorMessage());
  }

  public void testInvalidIntegerMessage() throws Exception {
    ArgsException e =
     new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                       'x', "Forty two");
    assertEquals("Argument -x expects an integer but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingIntegerMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER,
                                        'x', null);
    assertEquals("Could not find integer parameter for -x.", e.errorMessage());
  }

  public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
  }
}
```

代码清单14-15 ArgsException.java

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }


  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode, char errorArgumentId,
                       String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
```

```
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return String.format("Argument -%c unexpected.", errorArgumentId);
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                          errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                          errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                          errorArgumentId);
    case INVALID_DOUBLE:
      return String.format("Argument -%c expects a double but was '%s'.",
                          errorArgumentId, errorParameter);


    case MISSING_DOUBLE:
      return String.format("Could not find double parameter for -%c.",
                          errorArgumentId);
   }
   return "";
  }

  public enum ErrorCode {
   OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
   MISSING_STRING,
   MISSING_INTEGER, INVALID_INTEGER,
   MISSING_DOUBLE, INVALID_DOUBLE}
}
```

代码清单14-16 Args.java

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
   new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
   this.schema = schema;
   argsList = Arrays.asList(args);
   parse();
  }

  private void parse() throws ArgsException {
   parseSchema();
   parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
```

```
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                              elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                              elementId, null);
    }
  }

  private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                              argChar, null);
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
```

```
    return false;
  try {
   m.set(currentArgument);
   return true;
  } catch (ArgsException e) {
   e.setErrorArgumentId(argChar);
   throw e;
  }
}


public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
   return "-[" + schema + "]";
  else
   return "";
}

public boolean getBoolean(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
   b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
   b = false;
  }
  return b;
}

public String getString(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  try {
   return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
   return "";
  }
}

public int getInt(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  try {
   return am == null ? 0 : (Integer) am.get();
  } catch (Exception e) {
   return 0;
  }
}


public double getDouble(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  try {
   return am == null ? 0 : (Double) am.get();
  } catch (Exception e) {
   return 0.0;
  }
}
```

```
  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

对Args类所做的最主要的修改是在监测部分。从Args里面取出了大量代码，放到ArgsException中。很好。我们还把全部ArgumentMarshaler转移到了它们自己的文件中。更好！

优秀的软件设计，大都关乎分隔——创建合适的空间放置不同种类的代码。对关注面的分隔让代码更易于理解和维护。

特别有意思的是ArgsException中的errorMessage方法。显然，把错误信息格式化操作放在Args里面，违反了SRP原则。Args应该只处理参数，不该去管错误信息的格式。然而，把错误信息格式化代码放到ArgsException中是否有道理呢？

实话说，这是种折衷做法。不打算用ArgsException提供的错误信息的用户会想自己写错误信息。但如果有备好的错误信息，其方便之处也并非鲜见。

现在，显然我们已经非常接近本章开始处所展示的最终解决方案了。最后的工作留给你来练习完成。

14.4 小结

代码能工作还不够。能工作的代码经常会严重崩溃。满足于仅仅让代码能工作的程序员不够专业。他们会害怕没时间改进代码的结构和设计，我不敢苟同。没什么能比糟糕的代码给开发项目带来更深远和长期的损害了。进度可以重订，需求可以重新定义，团队动态可以修正。但糟糕的代码只是一直腐败发酵，无情地拖着团队的后腿。我无数次看到开发团队蹒跚前行，只因为他们匆匆搞出一片代码沼泽，从此之后命运再也不受自己控制。

当然，糟糕的代码可以清理。不过成本高昂。随着代码腐败下去，模块之间互相渗透，出现大量隐藏纠结的依赖关系。找到和破除陈旧的依赖关系又费时间又费劲。另一方面，保持代码整洁却相对容易。早晨在模块中制造出一堆混乱，下午就能轻易清理掉。更好的情况是，5分钟之前制造出混乱，马上就能很容易地清理掉。

所以，解决之道就是保持代码持续整洁和简单。永不让腐坏有机会开始。

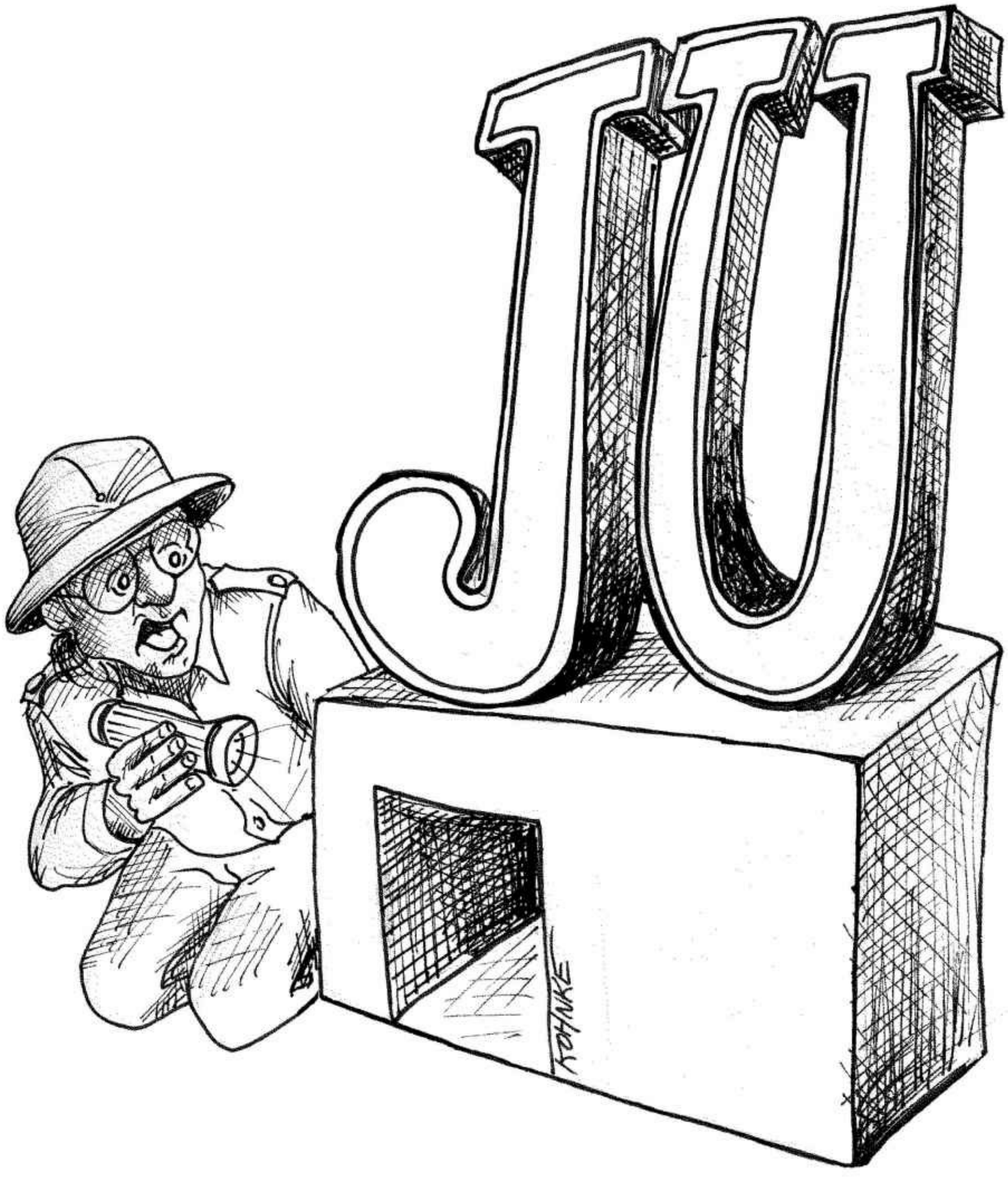
---

[1]
原注：最近我用Ruby语言重写了这个模块。大概只有Java版本的1/7大小，而且结构也稍好一些。

[2]
译注：即创建一个类型为ArgumentMarshaler的对象实体。

[3]
原注：为了避免这种情况发生，我添加了一个新的单元测试，调用所有FitNesse测试。

第15章　JUnit内幕



JUnit是最有名的Java框架之一。就像别的框架一样，它概念简单，定义精确，实现优雅。但它的代码是怎样的呢？本章将研判来自JUnit框架的一个

代码例子。

15.1　JUnit框架

JUnit有很多位作者，但它始于Kent Beck和Eric Gamma一次去亚特兰大的飞行旅程。Kent想学Java，而Eric则打算学习Kent的Smalltalk测试框架。“对于两个身处狭窄空间的奇客，还有什么会比拿出笔记本电脑开始编码来得更自然呢？[1]”经过3小时高海拔工作，他们写出了JUnit的基础代码。

我们要查看的模块，是用来帮忙鉴别字符串比较错误的一段聪明代码。该模块被命名为ComparisonCompactor。对于两个不同的字符串，例如ABCDE和ABXDE，它将用形如<...B[X]D...>的字符串来曝露两者的不同之处。

我可以做进一步解释，但测试用例会更有说服力。看看代码清单15-1，你将深入了解到该模块满足的需求。边看代码，边研究该测试的结构。它们能变得更简洁或更明确吗？

代码清单15-1　ComparisonCompactorTest.java

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;

```

```java
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact
("a");
    assertTrue("a expected:<[b]> but was:<[c]>".equals(failure))
;
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compa
ct(null);
    assertEquals("expected:<b[a]> but was:<b[c]>", failure);
  }

  public void testEndSame() {
    String  failure= new ComparisonCompactor(1, "ab", "cb").comp
act(null);
    assertEquals("expected:<[a]b> but was:<[c]b>", failure);
  }

  public void testSame() {
    String failure= new ComparisonCompactor(1, "ab", "ab").compa
ct(null);
    assertEquals("expected:<ab> but was:<ab>", failure);
  }

  public void testNoContextStartAndEndSame() {
    String failure= new ComparisonCompactor(0, "abc", "adc").com
pact(null);
    assertEquals("expected:<...[b]...> but was:<...[d]...>", fai
lure);
  }

  public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").com
pact(null);
    assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
  }

  public void testStartAndEndContextWithEllipses() {
    String failure=
      new ComparisonCompactor(1, "abcde", "abfde").compact(null)
```

```
;
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>",
 failure);
  }

  public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").comp
act(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
  }

  public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").comp
act(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
  }

  public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").comp
act(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
  }

  public void testComparisonErrorOverlapingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").co
mpact(null);
    assertEquals("expected:<...[]...> but was:<...[b]...>", fail
ure);
  }

  public void testComparisonErrorOverlapingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").co
mpact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
  }

  public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
    "abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[]...>", fail
ure);
  }

  public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
```

```
    new ComparisonCompactor(2, "abcdde", "abcde").compact(null
);
   assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", fail
ure);
  }

  public void testComparisonErrorWithActualNull() {
   String failure= new ComparisonCompactor(0, "a", null).compac
t(null);
   assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithActualNullContext() {
   String failure= new ComparisonCompactor(2, "a", null).compac
t(null);
   assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithExpectedNull() {
   String failure= new ComparisonCompactor(0, null, "a").compac
t(null);
   assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testComparisonErrorWithExpectedNullContext() {
   String failure= new ComparisonCompactor(2, null, "a").compac
t(null);
   assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testBug609972() {
   String failure= new ComparisonCompactor(10, "S&P500", "0").c
ompact(null);
   assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
  }
}
```

我对用到这些测试的ComparisonCompactor进行了代码覆盖分析。代码被100%覆盖了。每行代码、每个if语句和for循环都被测试执行了。于是我

对代码的工作能力有了极高的信心，也对代码作者们的技艺产生了极高的尊敬。

ComparisonCompactor的代码如代码清单15-2所示。

代码清单15-2　ComparisonCompactor.java（原始版本）

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;

  public ComparisonCompactor(int contextLength,
                             String expected,
                              String actual) {
    fContextLength = contextLength;
    fExpected = expected;
    fActual = actual;
  }

  public String compact(String message) {
    if (fExpected == null || fActual == null || areStringsEqual(
))
      return Assert.format(message, fExpected, fActual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(fExpected);
    String actual = compactString(fActual);
    return Assert.format(message, expected, actual);
```

```
  }

  private String compactString(String source) {
    String result = DELTA_START +
                        source.substring(fPrefix, source.length()
-
                          fSuffix + 1) + DELTA_END;
    if (fPrefix > 0)
      result = computeCommonPrefix() + result;
    if (fSuffix > 0)
      result = result + computeCommonSuffix();
    return result;
  }

  private void findCommonPrefix() {
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
      if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
        break;
    }
  }

  private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (;
         actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
           actualSuffix--, expectedSuffix--) {
      if (fExpected.charAt(expectedSuffix) != fActual.charAt(act
ualSuffix))
        break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
  }

  private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
             fExpected.substring(Math.max(0, fPrefix - fContextLe
ngth),
                                                fPrefix);
  }

  private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fConte
```

```
xtLength,
                         fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1,
 end) +
           (fExpected.length() - fSuffix + 1 < fExpected.length(
) -
               fContextLength ? ELLIPSIS : "");
  }

  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
  }
}
```

你可能会对这个模块有所抱怨。里面有些长表达式，有些奇怪的+1操作，如此等等。不过，总的来说，这个模块很不错。毕竟它原本可能被写成如代码清单15-3中的样子。

代码清单15-3　ComparisonCompator.java（背离版本）

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
```

```
      return Assert.format(msg, s1, s2);
    pfx = 0;

    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }

  private String compactString(String s) {
    String result =
      "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
      s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length
());
      result = result + (s1.substring(s1.length() - sfx + 1, end
) +
          (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : "
"));
    }
    return result;
  }
}
```

即便作者们把这个模块写得已经很棒，但童子军军规 [2] 却告诉我们，离时要比来时整洁。所以，我们怎样才能改进代码清单15-2中的原始代码

呢？

我首先看到的是成员变量的f前缀[N6]。在现今的运行环境中，这类范围性编码纯属多余。所以，先删除所有的f前缀。

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

下一步，在compact函数开始处，有一个未封装的条件判断[G28]。

```
public String compact(String message) {
 if (expected == null || actual == null || areStringsEqual())



   return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

这个条件判断应当封装起来，从而更清晰地表

达代码的意图。我们拆解出一个方法，解释这个条件判断。

```
public String compact(String message) {
  if (shouldNotCompact()



)
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {




  return expected == null || actual == null || areStringsEqual();





}
```

我也不太喜欢compact函数中的this.expected和this.actual符号。这个是我们把fExpected改为expected时发生的。为什么函数中的变量会与成员变量同名呢？它们不是该表示其他意思吗[N4]？我

们应该区分这些名称。

```
String compactExpected


 = compactString(expected


);
String compactActual


 = compactString(actual


);
```

否定式稍微比肯定式难理解一些[G29]。我们把if语句放到上头，调转条件判断。

```
public String compact(String message) {
  if (canBeCompacted()


) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
    return Assert.format(message, compactExpected, compactActual
);
  } else {
    return Assert.format(message, expected, actual);
  }
}

private boolean canBeCompacted()
```

```
 {
  return expected !

= null &&

 actual != null &&

 !areStringsEqual();
}
```

函数名很奇怪[N7]。尽管它的确会压缩字符串，但如果canBeCompact为false，它实际上就不会压缩字符串。用compact来命名，隐藏了错误检查的副作用。注意，该函数返回一条格式化后的消息，而不仅仅只是压缩后的字符串。所以，函数名其实应该是formatCompacted Comparison。在用以下参数调用时，读起来会好很多：

```
public String formatCompactedComparison(String message) {
```

两个字符串是在if语句体中压缩的。我们应当拆分出一个名为compactExpectedAndActual的方法。然而，我们希望formatCompactComparison函数完成所有的格式化工作。而compact…函数除了压

缩之外什么都不做[G30]。所以，做如下拆分：

```
...
 private String compactExpected;



  private String compactActual;



...
  public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
      compactExpectedAndActual();



      return Assert.format(message, compactExpected, compactActu
al);
    } else {
      return Assert.format(message, expected, actual);
    }
  }

  private void compactExpectedAndActual()


 {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
```

注意，这要求我们向成员变量举荐compactExpected和compactActual。我不喜欢新函数

最后两行返回变量的方式，但前两个可不是这样。它们没采用一以贯之的约定[G11]。我们应该修改findCommonPrefix和findCommonSuffix，返回前缀和后缀值。

```
private void compactExpectedAndActual() {
 prefixIndex


 = findCommonPrefix();
 suffixIndex


 = findCommonSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int


 findCommonPrefix() {
 int


 prefixIndex


 = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex


 < end; prefixIndex


++) {
    if (expected.charAt(prefixIndex


) != actual.charAt(prefixIndex
```

```
))
      break;
  }
  return prefixIndex;



}

private int


 findCommonSuffix() {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex


 && expectedSuffix >= prefixIndex;
      actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualS
uffix))
     break;
  }
  return


 expected.length() - expectedSuffix;
}
```

我们还应该修改成员变量的名称，使之更准确一点[N1]；毕竟它们都是索引。

仔细检查findCommonSuffix，其中藏了个时序性耦合 [G31]；它依赖于prefixIndex是由

findCommonSuffix计算得来的事实。如果这两个方法不是按这样的顺序调用，调试就会变得困难。为了暴露这个时序性耦合，我们将prefixIndex做成find的参数。

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix(prefixIndex


);
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonSuffix(int


 prefixIndex


) {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefix
Index;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualS
uffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

我对这样的方式不太满意。传递prefixIndex参数有些随意[G32]。它成功维持了执行次序，但对

于解释排序的需要却毫无作用。其他程序员可能会抹杀我们刚完成的工作，因为并没有迹象说明该参数确属必要。还是采取别的做法吧。

```
private void compactExpectedAndActual() {
 findCommonPrefixAndSuffix


();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private void


 findCommonPrefixAndSuffix()


 {
 findCommonPrefix


();
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (;
       actualSuffix >= prefixIndex && expectedSuffix >= prefixIn
dex;
       actualSuffix--, expectedSuffix--
  ) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualS
uffix))
      break;
  }
  suffixIndex = expected.length() - expectedSuffix;
}

private void


 findCommonPrefix() {
```

```
  prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++)
    if (expected.charAt(prefixIndex) != actual.charAt(prefixInde
x))
      break;
}
```

我们恢复findCommonPreffix和findCommonSuffix的原样，把findCommonSuffix的名称改为findCommonPrefixAndSuffix，让它在执行其他操作之前，先调用findCommonPrefix。这样一来，就以一种相比前种手段更为有效的方式建立了两个函数之间的时序关系。

```
private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int suffixLength = 1;
  for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
      break;
  }
  suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
  return s.charAt(s.length()-i);
}

private boolean suffixOverlapsPrefix(int suffixLength) {
  return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}
```

这样就好多了。它暴露出suffixIndex其实是后缀的长度，而且名字没取好。对于prefix也是如此。虽然在那样一种情形下index和length是同义的，但使用length一词却更有一贯性。问题在于，suffixIndex变量并不从0开始，它从1开始，所以并非真正的长度。这也是computeCommonSuffix中那些+1存在的原因[G33]。来修正它们吧。结果就是代码清单15-4。

代码清单15-4　ComparisonCompactor.java（过渡版本）

```
public class ComparisonCompactor {
  ...
  private int suffixLength;



  ...
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength


 = 0;



    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++)
{
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1
```

```
);
  }

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <=


 prefixLength ||
      expected.length() - suffixLength <=


 prefixLength;
  }
  ...
  private String compactString(String source) {
    String result =
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLen
gth


) +
      DELTA_END;
    if (prefixLength > 0)
      result = computeCommonPrefix() + result;
    if (suffixLength


 > 0)
      result = result + computeCommonSuffix();
    return result;
  }
  ...
  private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength


 +
        contextLength, expected.length()
    );
    return
    expected.substring(expected.length() - suffixLength
```

```
, end) +
   (expected.length() - suffixLength


 <
      expected.length() - contextLength ?
      ELLIPSIS : "");
 }
```

我们用charFromEnd中的那个-1替代了computeCommonSuffix中的一堆+1，前者更为合情合理，suffixOverlapsPrefix中的两个“<=”操作符也同理。这样我们就能修改suffixIndex和suffixLength的名称，极大地提升了代码的可读性。

不过还有一个问题。在消灭那些+1时，我注意到compactString中的以下代码：

```
if (suffixLength > 0)
```

看看代码清单15-4中的这行代码。因为suffixLength现在要比原本少1，我应该把“>”操作符改为“>=”操作符。那本无道理，不过现在 却有意义！这表示这么做没道理，而且可能是个缺陷。嗯，也不算是个缺陷。从之前的分析中我们可以看到，if语句现在会放置添加长度为零的后缀。在作出修改之前，if语句没有作用，因为suffixIndex永不

会小于1。

这说明compactString中的两个if语句都有问题！看起来它们都该删除。所以，我们将其注释掉，运行测试。测试通过了！那就重新构架compactString，删除没用的if语句，将函数改得更加简洁[G9]。

```
private String compactString(String source) {
  return
    computeCommonPrefix() +
    DELTA_START +
    source.substring(prefixLength, source.length() - suffixLengt
h) +
    DELTA_END +
    computeCommonSuffix();
}
```

这样就好多了！现在我们看到，compactString函数只是把片段组合起来。我们甚至可以让它更清晰。有许多细微的整理工作可做。与其拖着你遍历剩下的那些修改，我更愿意直接展示代码清单15-5中的结果。

代码清单15-5　ComparisonCompactor.java（最终版）

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
```

```
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
      int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual
);
  }

  private boolean shouldBeCompacted() {
    return !shouldNotBeCompacted();
  }

  private boolean shouldNotBeCompacted() {
    return expected == null ||
           actual == null ||
           expected.equals(actual);
  }

  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength)
```

```
      )
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
    expected.length() - suffixLength <= prefixLength;
  }

  private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
      if (expected.charAt(prefixLength) != actual.charAt (prefix
Length))
        break;
  }

  private String compact(String s) {
    return new StringBuilder()
    .append(startingEllipsis())
    .append(startingContext())
    .append(DELTA_START)
    .append(delta(s))
    .append(DELTA_END)
    .append(endingContext())
    .append(endingEllipsis())
    .toString();
  }

  private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
  }

  private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength)
;
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
  }
```

```
  private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
  }

  private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
      Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
  }

  private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
  }
}
```

这的确很漂亮。模块分解成了一组分析函数和一组合成函数。它们以一种拓扑方式排序，每个函数的定义都正好在其被调用的位置后面。所有的分析函数都先出现，而所有的合成函数都最后出现。

仔细阅读，你会发现我推翻了在本章较前位置做出的几个决定。例如，我将几个分解出来的方法重新内联为formatCompactComparison，我修改了souldNotBeCompacted表达式的意思。这种做法很常见。重构常会导致另一次推翻此次重构的重构。重构是一种不停试错的迭代过程，不可避免地集中于我们认为是专业人员该做的事。

15.2　小结

如此我们遵循了童子军军规。模块比我们发现它时更整洁了。不是说它原本不整洁。作者们做了卓越的工作。但模块都能再改进，我们每个人也有责任把模块改进得比发现时更整洁。

---

[1]　原注：*JUnit Pocket Guide*，Kent Beck，O’Reilly，2004，P.43。

[2]　原注：见前文有关章节。

第16章　重构SerialDate



如果你访问http://www.jfree.org/jcommon/index.php，就能找到JCommon类库。深入该类库，其中有个名为org.jfree.date的程序包。在该程序包中，有个名为SerialDate的类。我们即将剖析这个类。

SerialDate的作者是David Gilbert。David显然是位经验丰富、能力足够的程序员。如我们将看到的，他在代码中展示了极高的专业性和原则性。无

论怎么说，这都是“好代码”。而我将把它撕成碎片。

这并非恶意的行为。我也不认为自己比戴维强许多，有权对他的代码说三道四。其实，如果你看过我的代码，我敢说你也会发现好些该埋怨的东西。

不，这也并非傲慢无礼的行为。我所要做的，只是一种专业眼光的检视，不多也不少。那是我们都该坦然接受的做法。那是我们应该欢迎别人对自己做的事。只有通过这样的批评，我们才能学到东西。医生就是这样做的。飞行员就是这样做的。律师就是这样做的。我们程序员也需要学习如何这样做。

多说一句关于David Gilbert的事：David不止是位优秀的程序员。戴维有着将代码免费呈献给社区的勇气和好心。他公开代码，让所有人都能看到，邀请大众使用并审查。做得真好！

SerialDate（见代码清单B-1）是一个用Java呈现一个日期的类。为什么在Java已经有java.util.Date和java.util.Calendar的时候，还需要一个呈现日期的类呢？作者编写这个类，是为了响应我自己也常感到的痛苦。在开放的Javadoc（第67行）中，他很好

地解释了原因。我们可以质疑他的初衷，但我的确有处理这个问题的需要，而且我也欢迎有个关心日期甚于时间的类存在。

16.1　首先，让它能工作

在一个名为SerialDateTests的类（见代码清单B-2）中，有一些单元测试。测试都通过了。不幸的是，快览一遍测试，发现它们并没有测试所有东西[T1]。例如，用“查找使用”搜索方法MonthCodeToQuarter（第334行），会发现没有被用过[F4]。因此，单元测试并没有测试这个方法。

所以，我用Clover来检查单元测试覆盖了哪些代码。Clover报告说，在SerialDate的185个可执行语句中，单元测试只执行了91个（约50%）[T2]。覆盖图看起来像是一床满是补丁的棉被，整个类上布满大块的未执行代码。

我的目标是完整地理解和重构这个类。没有好得多的测试覆盖率，做不到这个。所以，我完全重起炉灶编写了自己的单元测试（见代码清单B-4）。

在阅读这些测试时，你可以看到，其中许多注释掉了。这些测试不能通过。它们代表了我以为SerialDate应该有的行为。在我重构SerialDate时，也将让这些测试通过。

即便有些测试被注释掉，Clover还是报告新的单元测试执行了185个可执行语句中的170个（92%）。这样就好多了，而且我想我们可以把这个数字提高些。

前几个注释掉的测试（第23～63行）是我一厢情愿。程序并没有设计为通过这些测试，但对我来说它们代表的行为显而易见[G2]。我不太确定testWeekdayCodeToString方法为何要写成那样，不过既然它已经在那儿，显然不该是区分大小写的。编写这些测试是区区小事[T3]，通过测试更加容易。我只修改了第259行和和263行，就能使用equalsIgnoreCase了。

我注释掉了第32行和第45行的测试，因为我不太明确是否应该支持tues和thurs缩写。

第153行和154行的测试不能通过。显然，它们本该通过[G2]。我们可以轻易地修正，只要对stringToMonthCode作出以下修改就行，对于第163行和213行的测试也一样。

```
457         if ((result < 1) || (result > 12)) {
                result = -1;
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equalsIgnoreCase(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
```

```
463                 if (s.equalsIgnoreCase(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
```

第318行注释掉的测试暴露了getFollowingDayOfWeek方法中的一个缺陷（第672行）。2004年12月25日是个周六。下一个周六是2005年1月1日。然而，运行测试时，会看到getFollowingDayOfWeek返回12月25日之后的周六还是12月25日。显然这不对[G3] [T1]。我们看到问题在第685行。那是个典型的边界条件错误[T5]。应该是这样：

```
685 if (baseDOW >= targetWeekday) {
```

很有意思，这个函数是之前一次修改的结果。修改记录（第43行）显示，getPreviousDayOfWeek、getFollowingDayOfWeek和getNearestDayOfWeek中的“缺陷”已被修正[T6]。

测试getNearestDayOfWeek（第705行）的单元测试testGetNearestDayOfWeek（第329行）之前的版本不像现在一样没有遗漏。我添加了大量测试用

例，因为初始的测试用例并没有全部通过[T6]。查看哪些测试用例被注释掉，你可以看到失败的模式，这很有启发。如果最近的日期是在未来，算法就会失败。显然存在某种边界条件错误[T5]。

Clover汇报的测试覆盖模式也很有趣[T8]。第719行根本没有执行！这意味着第718行的if语句总是得到false的结果。没错，看一眼代码就知道是这样。变量adjust总是为负，所以不会大于或等于4。所以，算法错了。

正确的算法如下所示：

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
  adjust -= 7;
return SerialDate.addDays(adjust, base);
```

最后，只要简单地抛出IllegalArgumentException异常而不是从weekInMonthToString和relativeToString返回错误字符串，第417行和429行的测试也能通过。

做出这些修改后，所有的单元测试都通过了，我确信SerialDate现下可以工作。是时候让它“做

对”了。

16.2　让它做对

我们将从头到尾遍历SerialDate，同时加以改进。尽管在本章的讨论中你看不到这个过程，在每次做修改后，我还是要运行全部JCommon单元测试，包括我为SerialDate改进的那些单元测试。所以，后面你看到的所有修改，对于JCommon都是可工作的。

从第1行开始，我看到大量有关许可、版权、作者和修改历史的注释。我明白，的确有些法律事宜要说明，所以版权和许可信息应该保留。另外，修改历史是产生于19世纪60年代的古董，现今源代码控制工具可以帮我们做到这个。应该删掉修改历史[C1]。

从第61行开始的导入列表应该通过使用java.text.*和java.util.*来缩短。[J1]

Javadoc的HTML格式化工作（第67行）令我畏惧。一个源文件里面有多种语言，我有点发怵。这条注释有4种语言：Java、英文、Javadoc和html[G1]。有那么多语言，就很难直截了当。例如，生成Javadoc后，第71行和72行原本很好的位置就丢失了，而且谁想在源代码中看到<ul>和<li>这

样的东西呢？更好的策略可能是用<pre>标签把整个注释部分包围起来，这样，对于源代码的格式化只会限于Javadoc之内[1]。

第86行是类声明。这个类为何要命名为SerialDate？Serial一词有什么妙处吗？是不是因为该类派生自Serializable？看来不是这样的。

别猜了，我知道为什么（或者我认为自己知道）何以要用Serial一词。线索就在位于第98行和101行的常量SERIAL LOWER BOUND和SERIAL UPPER BOUND。更好的线索在从第830行开始的注释中。该类被命名为SerialDate，是因为它用“序列数”（serial number）来实现，该系列数恰好是从1899年12月30日后的天数。

对此我有两个问题。首先，术语“序列数”并不真对。可能有点诡辩，但其呈现方式却更接近相对偏移甚于序列数。术语“序列数”更多地用于产品版本标识，而非日期标识。我没发现这个名称特别有描述力[N1]。更有描述力的术语大概是“顺序”（ordinal）。

第二个问题更突出。名称SerialDate暗示了一种实现。该类是个抽象类。没必要暗示任何有关实现的事。实际上，没理由隐藏实现！我发现这个名称

放在了不正确的抽象层级上[N2]。以我之见，该类的名称应该就是简单的Date。

不幸的是，Java类库里面有太多叫Date的类了，所以这大概也不是最好的名称。因为这个类是关于日期而非时间，我想将其命名为Day，但这个名字也在多处被滥用。最后，我选了DayDate作为最佳折衷方案。

从现在起，我将使用术语DayDate。请记住，你读到的代码清单，还是用的SerialDate。

我理解为何DayDate继承自Comparable和Serializable。不过，为什么它要继承自MonthConstants呢？类MonthConstants（见代码清单B-3）只是一大堆定义了月份的静态常量。从常量类继承是Java程序员用的一种老花招，这样他们就能避免形如MonthConstants.January的表达式，不过这是个坏主意[J2]。MonthConstants其实应该是个枚举。

```
public abstract class DayDate implements Comparable,
                                         Serializable {
  public static enum Month {
    JANUARY(1),
    FEBRUARY(2),
    MARCH(3),
    APRIL(4),
    MAY(5),
    JUNE(6),
```

```
    JULY(7),
    AUGUST(8),
    SEPTEMBER(9),
    OCTOBER(10),
    NOVEMBER(11),
    DECEMBER(12);

    Month(int index) {
      this.index = index;
    }

    public static Month make(int monthIndex) {
      for (Month m : Month.values()) {
        if (m.index == monthIndex)
          return m;
      }
      throw new IllegalArgumentException("Invalid month index "
+ monthIndex);
    }
    public final int index;
  }
```

把MonthConstants改成枚举，导致对DayDate类和用到这个类的代码的一些修改。我花了一个小时来改代码。不过，原来以int为月份类型的函数，现在都用上Month枚举元素了。这意味着我们可以去除isValidMonthCode方法（第326行），以及monthCodeToQuarter等位置的月份代码错误检查（第356行）了[G5]。

下一步，我们看到第91行，serialVersionUID。该变量用于控制序列号。如果我们修改了它，用这个软件编写的旧版本DayDate都将不再可用，而是返回一个InvalidClassException异常。如果你没有声

明serialVersionUID变量，则编译器会自动生成一个，每次修改模块时都会得到不一样的值。我知道，所有的文档都建议手工控制这个变量，但对我来说自动控制序列号安全得多[G4]。我宁肯调试InvalidClassException，也不愿意见到如果忘记修改serialVersionUID引起的后续工作。所以，我要删除这个变量——至少暂时这么做[2]。

我发现第93行的注释是多余的。这正是谎言和误导信息所在之地[C2]。所以我要干掉它和它的同类。

第97行和100行的注释有关序列数，我之前已经讨论过这个问题[C1]。它们描述的变量是DayDate能够描述的最早和最后的日期。这可以搞得更清楚些[N1]。

```
public static final int EARLIEST_DATE_ORDINAL = 2;    // 1/1/19
00
public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31
/9999
```

我不太清楚为什么EARLIEST_DATE_ORDINAL是2而不是0。在第829行的注释中有个提示，说明这与用Microsoft Excel展示日期的方式有关。在DayDate的派生类

SpredsheetDate中能看得更深入（见代码清单B-5）。第71行的注释很好地描述了这个问题。

我的问题是，这看来应该与SpreadsheetDate有关，与DayDate无关才对。所以，EARLIEST_DATE_ORDINAL和LATEST_DATE_ORDINAL实在不该属于DayDate，应该移到SpreadSheeDate中[G6]。

的确，搜索一下代码就知道，这些变量值仅在SpreadSheetDate中用到。DayDate中没用到，JCommon框架的其他类中也没有用。所以，我将把它们向下移到SpreadSheetDate中。

下面两个变量，MINIMUN_YEAR_SUPPORTED和MAXIMUM_YEAR_SUPPORTED（第104行和107行）地位尴尬。显然，如果DayDate是个没有提供实现铺垫的抽象类，它就不该告知我们有关最小和最大年份的信息。同样，我很想把这些变量向下移到SpreadSheetDate中[G6]。然而，快速查找这些变量的使用情况，会发现另一个类也在用：RelativeDayOfWeekRule（见代码清单B-6）。在第177行和178行，getDate函数中，它们被用来检查getDate的年份参数是否有效。抽象类的用户需要得知其实现信息，这是个矛盾。

我们要做的是既提供信息，又不污染DayDate。通常，我们会从派生类实体中获取实现信息。不过，并未向getDate函数传入DayDate的实体，反而返回了这么一个实体。这意味着必须在某处创建实体。第187～205行提供了线索。DayDate实体是在getPreviousDayOfWeek、getNearestDayOfWeek或getFollowingDayOfWeek这三个函数其中之一里面创建的。看回DayDate代码清单，我们看到，这些函数（第638～724行）全都返回了由addDays（第571行）创建的日期实体，addDays调用CreateInstance（第808行），创建出一个SpreadSheetDate！[G7]。

通常来说，基类不宜了解其派生类的情况。为了修正这个毛病，我们应该利用抽象工厂模式（ABSTRACT FACTORY）[3]，创建一个DayDateFactory。该工厂将创建我们所需要的DayDate的实体，并回答有关实现的问题，例如最大和最小日期之类。

```
public abstract class DayDateFactory {
  private static DayDateFactory factory = new SpreadsheetDateFac
tory();
  public static void setInstance(DayDateFactory factory) {
    DayDateFactory.factory = factory;
  }
  protected abstract DayDate _makeDate(int ordinal);
  protected abstract DayDate _makeDate(int day, DayDate.Month mo
nth, int year);
  protected abstract DayDate _makeDate(int day, int month, int y
```

```
ear);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();

  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }

  public static DayDate makeDate(int day, DayDate.Month month, i
nt year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
  }

  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }

  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

该工厂类用makeDate方法替代了createInstance方法，前者的名称稍好一些[N1]。在初始状态下，它使用SpreadsheetDateFactory，但随时可以使用其他工厂。委托到抽象方法的静态方法混合采用了单件模式（SINGLETON）、油漆工模式 [4] 和抽象工厂模式 [5] ，我发现这种手段很有用。

SpreadsheetDateFactory看起来像这个样子：

```
public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, DayDate.Month month, int yea
r) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
        calendar.get(Calendar.DATE),
        DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
        calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

如你所见，我已经把MINIMUM_YEAR_SUPPORTED和MAXIMUM_YEAR_SUPPORTED变量移到了它们该在的SpreadsheetDate中[G6]。

DayDate的下一个问题是第109行的日期常量。这些常量其实应该是枚举[J3]。我们之前见过这种模式，不再赘述。你可以在最终的代码清单中看到。

跟着，我们看到第140行一系列以LAST_DAY_OF_MONTH开头的数组。首先，描述这些数组的注释全属多余[C3]。光看名称就够了。所以我要删除这些注释。

这个数组没理由不是私有的[G8]，因为有个静态函数lastDayOfMonth提供同样的数据。

下一个数组AGGREGATE_DAYS_TO_END_OF_MONTH更神秘一些，在JCommon框架中根本没用到它[G9]。所以我直接删除了。

对于LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_也一样。

AGGREGATE_DAYS_TO_END_OF_PRECEDI只在SpreadsheetDate中用到（第434行和473行）。是否把它移到SpreadsheetDate中去是个问题。不转移的理由是，该数组并不专属于任何特定的实现

[G6]。另一方面，实际上并不存在SpreadsheetDate之外的实现，所以，数组应该移到靠近其使用位置的地方[G10]。

说服我的理由是保持一致[G11]，数组应该私有，并通过类似julianDateOfLastDayOfMonth这样的函数来暴露。看来没人需要那样的函数。而且，如果有新的DayDate实现需要该数组，可以轻易地把它移回到DayDate中去。所以我就把它移到SpreadsheetDate里面了。

对于LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_也一样。

跟着，我们看到三组可以转换为枚举的常量（第162～205行）。第一个用来选择月份中的一周。我将其转换为名为WeekInMonth的枚举。

```
public enum WeekInMonth {
  FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
  public final int index;

  WeekInMonth(int index) {
    this.index = index;
  }
}
```

第二组常量（第177～187行）有点麻烦。INCLUDE_NONE、INCLUDE_FIRST、INCLUDE_SECOND和INCLUDE_BOTH常量用于描述某个范围的终止日是否包含在该范围之内。数学上，用术语“开放区间”、“半开放区间”和“闭合区间”来表示。我想，用数学术语来命名会更清晰[N3]，所以就将其转换为枚举DateInterval，其中包括CLOSED、CLOSED_LEFT、CLOSED_RIGHT和OPEN枚举元素。

第三组常量（第18～205行）描述了是否该在最后、下一个或最近的日期实体中呈现对某个星期的特定一天的查找结果。怎么命名是个难题。最终，我给WeekdayRange设定了LAST、NEXT和NEAREST枚举元素。

你也许不会同意我取的名字。对我而言这些名字有意义，但对你可能就不然。要点是它们眼下变成了易于修改的形式[J3]。不再以整数形式传递，而是作为符号传递。我可以用IDE的“修改名称”功能来改动名称或类型，无需担忧漏掉代码中某处-1或2之类的数字，也不必担忧某些int参数声明处于描述不佳的状态。

第208行的描述字段看来没有任何地方用到。我把它及其取值器和赋值器都删掉了。

我还删除了第213行的默认构造器[G12]。编译器会为我们自动生成的。

略过isValidWeekdayCode方法（第216～238行），在创建Day枚举时已经把它删掉了。

于是来到stringToWeekdayCode方法（第242～270行）。没有方法签名增添价值的Javadoc都是废话[C3]、[G12]，唯一的价值是对返回值−1的描述。然而，因为我们改用了Day枚举，这条注释就完全错误了[C2]。该方法现在抛出一个IllegalArgumentException异常。所以我删除了Javadoc。

我还删除了参数和变量声明中的全部final关键字。我敢说，它们毫无价值，空自混淆视听惑[G12]。删除这些final，不合某些成例。例如，Robert Simmons [6] 就强烈建议我们"……在代码中遍布final。"我不能苟同。我认为，final有少数的好用法，例如偶尔使用的final常量，但除此之外该关键字利小于弊。我这么认为，或许是因为final可能捕获到的那些错误类型，早已被我编写的单元测试捕获了。

我不喜欢for循环（第259行和263行）中的那些if语句[G5]，所以我利用"||"操作符把它们连接为单

个if语句。我还使用Day枚举整理for循环，做了一些装饰性的修改。

我认为，这个方法并不真属于DayDate类。它其实是Day的一个解析函数。所以，我将它移到Day枚举中。不过，那样Day枚举就会变得太大。因为Day的概念并不依赖于DayDate，我就把Day枚举移到DayDate类之外，放到它自己的源代码文件中。

我还把下一个函数，weekdayCodeToString（第272～286行），移植到Day枚举中，称其为toString。

```
public enum Day {
  MONDAY(Calendar.MONDAY),
  TUESDAY(Calendar.TUESDAY),
  WEDNESDAY(Calendar.WEDNESDAY),s
  THURSDAY(Calendar.THURSDAY),
  FRIDAY(Calendar.FRIDAY),
  SATURDAY(Calendar.SATURDAY),
  SUNDAY(Calendar.SUNDAY);

  public final int index;
  private static DateFormatSymbols dateSymbols = new DateFormatS
ymbols();

  Day(int day) {
    index = day;
  }

  public static Day make(int index) throws IllegalArgumentExcept
ion {
    for (Day d : Day.values())
      if (d.index == index)
```

```
      return d;
    throw new IllegalArgumentException(
        String.format("Illegal day index: %d.", index));
  }

  public static Day parse(String s) throws IllegalArgumentExcept
ion {
    String[] shortWeekdayNames =
      dateSymbols.getShortWeekdays();
    String[] weekDayNames =
      dateSymbols.getWeekdays();

    s = s.trim();
    for (Day day : Day.values()) {
      if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
          s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
      }
    }
    throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
  }

  public String toString() {
    return dateSymbols.getWeekdays()[index];
  }
}
```

有两个getMonth函数（第288～316行）。第一个函数调用第二个函数。第二个函数只被第一个函数调用。所以，我把这两个函数合二为一，而且极大地简化之[G9][G12][F4]。最后，我把名称修改得更具自我描述力[N1]。

```
public static String[] getMonthNames() {
  return dateFormatSymbols.getMonths();
}
```

由于有了Month枚举，函数isValidMonthCode（第326～346行）就变得没什么用，所以我把它删除了[G9]。

函数monthCodeToQuarter（第356～375行）有特性依恋（FEATURE ENVY）[7] 的味道，可以是Month枚举中的一个名为quarter的方法，我就这么办了。

```
public int quarter() {
  return 1 + (index-1)/3;
}
```

这样一来，Month枚举就大到需要放到自己的类中了。我把它从DayDate中移出来，与Day枚举保持一致[G11][G13]。

下两个方法被命名为monthCodeToString（第377～426行）。我们再次看到其中一个方法使用标识调用其兄弟方法的模式。将标识作为参数传递给函数的做法通常不太好，尤其是当该标识只是有关其输出格式时[G15]。我重命名、简化、重新构架了这些函数，并把它们移到Month枚举中[N1][N3][G14]。

```
public String toString() {
  return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
  return dateFormatSymbols.getShortMonths()[index - 1];
}
```

下一个方法是stringToMonthCode（第428～472行）。我重新为它命名，转移到Month枚举中，并且简化之[N1][N3][C3][G14][G12]。

```
public static Month parse(String s) {
  s = s.trim();
  for (Month m : Month.values())
    if (m.matches(s))
      return m;

  try {
    return make(Integer.parseInt(s));
  }
  catch (NumberFormatException e) {}
  throw new IllegalArgumentException("Invalid month " + s);
}

private boolean matches(String s) {
  return s.equalsIgnoreCase(toString()) ||
         s.equalsIgnoreCase(toShortString());
}
```

方法isLeapYear（第495～517行）可以写得更具表达力一些[G16]。

```
public static boolean isLeapYear(int year) {
```

```
  boolean fourth = year % 4 == 0;
  boolean hundredth = year % 100 == 0;
  boolean fourHundredth = year % 400 == 0;
  return fourth && (!hundredth || fourHundredth);
}
```

下一个函数leapYearCount（第519～536行）并不真属于DayDate。除了SpreadsheetDate中的两个方法外，没有其他调用者。所以我将它往下放。

函数lastDayOfMonth（第538～560行）使用了LAST_DAY_OF_MONTH数组。该数组应该隶属于Month枚举[G17]，所以我就把它移到那儿去了。我还简化了这个函数，使其更具表达力[G16]。

```
public static int lastDayOfMonth(Month month, int year) {
  if (month == Month.FEBRUARY && isLeapYear(year))
    return month.lastDay() + 1;
  else
    return month.lastDay();
}
```

现在，事情变得比较有趣一些了。下一个函数是addDays（第562～576行）。首先，由于该函数对DayDate的变量进行操作，它就不该是静态的[G18]。所以，我把它修改为实体方法。其次，它调用了函数toSerial。这个函数应该重新命名为toOrdial [N1]。最后，该方法可以简化。

```
public DayDate addDays(int days) {
  return DayDateFactory.makeDate(toOrdinal() + days);
}
```

对于addMonth（第578～602行）也一样。它应该是个实体方法[G18]。算法太过复杂，所以我利用解释临时变量模式（EXPLAINING TEMPORARY VARIABLES）[8] 来使其更为透明。我还将方法getYYY重命名为getYear [N1]。

```
public DayDate addMonths(int months) {
  int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1
;
  int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
  int resultYear = resultMonthAsOrdinal / 12;
  Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
  int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultY
ear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth
);
  return DayDateFactory.makeDate(resultDay, resultMonth, resultY
ear);
}
```

对于函数addYear（第604～626行）也照方办理。

```
public DayDate plusYears(int years) {
  int resultYear = getYear() + years;
  int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), re
sultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResu
```

```
ltYear);
  return DayDateFactory.makeDate(resultDay, getMonth(), resultYe
ar);
}
```

把这些方法从静态方法变为实体方法，让我有点心头发痒。用date.addDays(5)这样的表达方法，是不是明确地表示了date对象并没变动以及返回了一个DayDate的新实体呢？或者，它只是错误地暗示我们往date对象添加了5天呢？你可能不会认为这是个大问题，但下列代码却可能会有欺骗性。

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // bump date by one week.
```

有些读到这段代码的人会认为addDays在修改date对象。所以，我们需要消除这种歧义的名称[N4]。我把名称改为plusDays和plusMonths。我认为，方法的初衷很清楚地被

```
DayDate date = oldDate.plusDays(5);
```

所体现，不过下列代码对认为date对象被修改的读者来说，看起来并不那么顺畅：

```
date.plusDays(5);
```

算法越来越有趣，getPreviousDayOfWeek（第628～660行）可以工作，不过有点复杂了。经过一番思考，了解到它的功能后[G21]，我就能够使用解释临时变量模式来简化它[G19]，使其更为清晰。我还将它从静态方法改为实体方法[G18]，并删除了重复的实体方法[G5]（第997～1008行）。

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().in
dex;
  if (offsetToTarget >= 0)
    offsetToTarget -= 7;
  return plusDays(offsetToTarget);
}
```

对getFollowingDayOfWeek（第662～693行）也如法炮制：

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().in
dex;
  if (offsetToTarget <= 0)
    offsetToTarget += 7;
  return plusDays(offsetToTarget);
}
```

下一个函数是我们之前修改过的getNearestDayOfWeek（第695～726行）。我之前所做的修改和前两个函数没有保持一致[G11]。所以我将它改得和这两个函数保持一致，并且使用解释临时变量模式[G19]来阐明算法。

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
  int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek()
.index;
  int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
int offsetToPreviousTarget = offsetToFutureTarget - 7;

  if (offsetToFutureTarget > 3)
    return plusDays(offsetToPreviousTarget);
  else
    return plusDays(offsetToFutureTarget);
}
```

方法getEndOfCurrentMonth（第728～740行）有点奇怪，因为它获取了DayDate参数，从而成为一个依恋[G14]其自身类的实体方法。我将其改为真正的实体方法，并修改了几个名称。

```
public DayDate getEndOfMonth() {
  Month month = getMonth();
  int year = getYear();
  int lastDay = lastDayOfMonth(month, year);
  return DayDateFactory.makeDate(lastDay, month, year);
}
```

重构weekInMonthToString（第742～761行）的过程非常有趣。利用IDE的重构工具，我先将其移到我之前创建的WeekInMonth枚举中，再将其重命名为toString。跟着，我把它从静态方法改为实体方法。所有的测试都通过了。（你能猜出来我打算做什么吗？）

接下来，我删掉了整个方法！有5个断言失败了（第411～415行，代码清单B-4）。我改动了这些代码行，让它们使用枚举元素的名称（FIRST、SECOND……）。全部测试都通过了。你知道为什么吗？你能否知道为什么这些步骤都是必要的吗？重构工具确保之前对weekInMonthToString方法的调用现在都调用weekInMonth枚举元素的toString方法，全部枚举元素都以返回其名称的形式实现了toString方法……

我不幸有点聪明过头了。这一套美妙的重构下来，我终于意识到，这个函数的唯一调用者，就是我刚修改的测试，所以我删除了这些测试。

愚我一次，是你之耻。愚我两次，是我之耻！所以，在判定除了测试之外没有人调用过relativeToString（第765～781行）后，我就删除了该函数及其测试。

我们最后将其改为这个抽象类的抽象方法。第一个函数保持了原样：toSerial（第838～844行）。前文我曾把名称改为toOrdinal。以现在的情形看，我决定应该把名称改为getOrdinalDay。

下一个抽象方法是toDate（第838～844行）。它将DayDate转换为java.util.Date。这个方法为何是抽象的？查看其在SpreadsheetDate中的实现（第198～207行，代码清单B-5），可以看到它并不依赖于该类的实现[G6]。所以，我把它往上推了。

方法getYYYY、getMonth和getDayOfMonth已经是抽象方法。不过，getDayOfWeek方法是另一个应该从SpreadsheetDate中提出来的方法，因为它不依赖于DayDate之外的东西[G6]。是这样吗？

仔细阅读（第247行，代码清单B-5），可以发现该算法暗中依赖于顺序日期的起点（换言之，第0天的星期日数）。所以，即便该方法没有物理上的依赖，也不能移到DayDate中，因为它的确有逻辑上的依赖。

这样的逻辑依赖困扰了我[G22]。如果有什么东西在逻辑上依赖实现的话，也该有什么物理上的依赖存在。我也认为，算法本身也该有一小部分依赖于实现。

所以我在DayDate中创建了一个名为getDayOfWekForOrdinalZero的抽象方法，并在SpreadsheetDate中实现它，返回Day.SATURDAY。然后我把getDayOfWeek上移到DayDate中，并调用getOrdinalDay和getDayOfWeekForOrdinal Zero。

```
public Day getDayOfWeek() {
  Day startingDay = getDayOfWeekForOrdinalZero();
  int startingOffset = startingDay.index - Day.SUNDAY.index;
  return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

顺便说一句，请仔细阅读第895～899行的注释。这样的重复有必要吗？通常，我会删除这类注释。

下一个方法是compare（第902～913行）。同样，该抽象方法是不恰当的[G6]。我将其实现上移到DayDate。其名称也不足够有沟通意义[N1]。方法实际上返回的是自参数日期以来的天数，所以我把名称改为daysSince。我还注意到该方法没有测试，就为它编写了测试。

下面6个函数（第915～980行）全都是应该在DayDate中实现的抽象方法。我把它们全都从SpreadsheetDate中抽出来了。

最后一个函数isInRange（第982～995行）也需要推到上一层并重构之。那个switch语句有点丑陋[G23]，可以把那些条件判断移到DateInterval枚举中去。

```
public enum DateInterval {
  OPEN {
    public boolean isIn(int d, int left, int right) {
      return d > left && d < right;
    }
  },
  CLOSED_LEFT {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d < right;
    }
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };

  public abstract boolean isIn(int d, int left, int right);
}


public boolean isInRange(DayDate d1, DayDate d2, DateInterval i
nterval) {
  int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
  return interval.isIn(getOrdinalDay(), left, right);
}
```

我们来到了DayDate的末尾。现在我们要从头到尾再过一次，看看整个重构过程是怎样良好执行的。

首先，开端注释过时已久，我缩短并改进了它[C2]。

然后，我把全部枚举移到它们自己的文件中[G12]。

跟着，我把静态变量（dateFormatSymbols）和3个静态方法（getMonthNames、isLeapYear和lastDayOfMonth）移到名为DateUtil的新类中[G6]。

我把那些抽象方法上移到它们该在的顶层类中[G24]。

我把Month.make改为Month.fromInt [N1]，并如法炮制所有其他枚举。我还为全部枚举创建了toInt( )访问器，把index字段改为私有。

在plusYears和plusMonths中存在一些有趣的重复[G5]，我通过抽离出名为correctLastDayOfMonth的新方法消解了重复，使这3个方法清晰多了。

我消除了魔术数1 [G25]，用Month.JANUARY.toInt( )或Day.SUNDAY.toInt( )做

了恰当的替换。我在SpreadsheetDate上花了点时间，清理了一下算法。最终结果在代码清单B-7～16中。

有趣的是，DayDate的代码覆盖率降低 到了84.9%！这并不是因为测试到的功能减少了，而是因为该类缩减得太多，导致少量未覆盖到的代码行拥有了更大权重。DayDate的53个可执行语句中有45个得到测试覆盖。未覆盖的代码行微细到不值得测试。

16.3　小结

我们再一次遵从了童子军军规。我们签入的代码，要比签出时整洁了一点。虽然花了点时间，不过很值得。测试覆盖率提升了，修改了一些缺陷，代码清晰并缩短了。后来者有望比我们更容易地应付这些代码。他也有可能把代码整理得更干净些。

16.4　文献

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[Simmons04]：*Hardcore Java* , Robert Simmons, Jr., O’Reilly, 2004.

[Refactoring]：*Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

[Beck97]：*Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

---

[1]　原注：更好的解决方案是让Javadoc不对注释做格式化，这样注释在代码和文档中就会是一种样式。

[2]　原注：本章的好几个审读者都不这么认为。他们主张，在开源框架中，手工控制序列ID会比较好，因为较小的修改不会导致序列化后的日期无效。这是种中肯的观点。然而，尽管会不方便，但

失败就会有个清晰的原因。另一方面，如果该类的作者忘记更新序列ID，则失败模式就会不可预期，而且可能会隐藏得很深。我认为，这个故事的精髓在于，不应该跨版本做反序列化处理。

[3] 原注：[GOF]。

[4] 原注：Ibid。

[5] 原注：Ibid。

[6] 原注：[Simmons04], p. 73。

[7] 原注：[Refactoring]。

[8] 原注：[Beck97]。

第17章　味道与启发



Martin Fowler在其妙书*Refectoring:Improving the Design of Existing Code* [1] 中指出了许多不同的“代码味道”。下面的清单包括很多Martin提出的味道，还添加了更多我自己提出的，也包括我借以历练本业的其他珍宝与启发。

我藉由遍览和重构几个不同的程序总结出这个

清单。每次修改，我都问自己为什么要这样改，把修改的原因写下来。结果就是得到相当长的清单，给出在读代码时让我闻起来不舒服的味道。

清单应按顺序阅读，并作为一种参考来使用。

17.1　注释

C1：不恰当的信息

让注释传达本该更好地在源代码控制系统、问题追踪系统或任何其他记录系统中保存的信息，是不恰当的。例如，修改历史记录只会用大量过时而无趣的文本搞乱源代码文件。通常，作者、最后修改时间、SPR数等元数据不该在注释中出现。注释只应该描述有关代码和设计的技术性信息。

C2：废弃的注释

过时、无关或不正确的注释就是废弃的注释。注释会很快过时。最好别编写将被废弃的注释。如果发现废弃的注释，最好尽快更新或删除掉。废弃的注释会远离它们曾经描述的代码，变成代码中无关和误导的浮岛。

C3：冗余注释

如果注释描述的是某种充分自我描述了的东西，那么注释就是多余的。例如：

```
i++; // increment i
```

另一个例子是除函数签名之外什么也没多说（或少说）的Javadoc：

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
   throws ManagedComponentException
```

注释应该谈及代码自身没提到的东西。

C4：糟糕的注释

值得编写的注释，也值得好好写。如果要编写一条注释，就花时间保证写出最好的注释。字斟句酌。使用正确的语法和拼写。别闲扯，别画蛇添足，保持简洁。

C5：注释掉的代码

看到被注释掉的代码会令我抓狂。谁知道它有多旧？谁知道它有没有意义？没人会删除它，因为大家都假设别人需要它或是有进一步计划。

那样的代码就这样腐烂掉，随着时间推移，越

来越与系统没关系。它调用不复存在的函数。它使用已改名的变量。它遵循已被废弃的约定。它污染了所属的模块，分散了想要读它的人的注意力。注释掉的代码纯属厌物 。

看到注释掉的代码，就删除它！别担心，源代码控制系统还会记得它。如果有人真的需要，可以签出较前的版本。别被它搞到死去活来。

17.2 环境

E1：需要多步才能实现的构建

构建系统应该是单步的小操作。不应该从源代码控制系统中一小点一小点签出代码。不应该需要一系列神秘指令或环境依赖脚本来构建单个元素。不应该四处寻找额外的小JAR、XML文件和其他系统所需的杂物。你应当 能够用单个命令签出系统，并用单个指令构建它。

```
svn get mySystem
cd mySystem
ant all
```

E2：需要多步才能做到的测试

你应当能够发出单个指令就可以运行全部 单元测试。能够运行全部测试是如此基础和重要，应该快速、轻易和直截了当地做到。

17.3　函数

F1：过多的参数

函数的参数量应该少。没参数最好，一个次之，两个、三个再次之。三个以上的参数非常值得质疑，应坚决避免。（参见前文“函数参数”一节。）

F2：输出参数

输出参数违反直觉。读者期望参数用于输入而非输出。如果函数非要修改什么东西的状态不可，就修改它所在对象的状态好了。（参见前文“输出参数”一节。）

F3：标识参数

布尔值参数大声宣告函数做了不止一件事。它们令人迷惑，应该消灭掉。（参见前文“标识参数”一节。）

F4：死函数

永不被调用的方法应该丢弃。保留死代码纯属

浪费。别害怕删除函数。记住，源代码控制系统还会记得它。

17.4　一般性问题

G1：一个源文件中存在多种语言

当今的现代编程环境允许在单个源文件中存在多种不同语言。例如，Java源文件可能还包括XML、HTML、YAML、JavaDoc、英文、JavaScript等语言。另例，JSP文件可能还包括HTML、Java、标签库语法、英文注释、Javadoc、XML、JavaScript等。往好处说是令人迷惑，往坏处说就是粗心大意、驳杂不精。

理想的源文件包括且只包括一种语言。现实上，我们可能会不得不使用多于一种语言。但应该尽力减少源文件中额外语言的数量和范围。

G2：明显的行为未被实现

遵循“最小惊异原则”（The Principle of Least Surprise）[2]，函数或类应该实现其他程序员有理由期待的行为。例如，考虑一个将日期名称翻译为表示该日期的枚举的函数。

```
Day day = DayDate.StringToDay(String dayName);
```

我们期望字符串Monday翻译为Day.MONDAY。我们也期望常用缩写形式也能被翻译出来，我们还期待函数忽略大小写。

如果明显的行为未被实现，读者和用户就不能再依靠他们对函数名称的直觉。他们不再信任原作者，不得不阅读代码细节。

G3：不正确的边界行为

代码应该有正确行为，这话看似明白。问题是我们很少能明白正确行为有多复杂。开发者常常写出他们以为能工作的函数，信赖自己的直觉，而不是努力去证明代码在所有的角落和边界情形下真能工作。

没什么可以替代谨小慎微。每种边界条件、每种极端情形、每个异常都代表了某种可能搞乱优雅而直白的算法的东西。别依赖直觉 。追索每种边界条件，并编写测试。

G4：忽视安全

切尔诺贝利核电站崩塌了，因为电厂经理一条又一条地忽视了安全机制。遵守安全就不便于做试验。结果就是试验未能运行，全世界都目睹首个民用核电站大灾难。

忽视安全相当危险。手工控制serialVersionUID可能有必要，但总会有风险。关闭某些编译器警告（或者全部警告！）可能有助于构建成功，但也存在陷于无穷无尽的调试的风险。关闭失败测试、告诉自己过后再处理，这和假装刷信用卡不用还钱一样坏。

G5：重复

有一条本书提到的最重要的规则之一，你应该非常严肃地对待。实际上，每位编写有关软件设计的作者都提到这条规则。Dave Thomas和Andy Hunt称之为DRY原则（Don’t Repeat Yourself，别重复自己）[3]。Kent　Beck将它列为极限编程核心原则之一，并称之为“一次，也只一次”。Ron Jeffries将这条规则列在第二位，地位只低于通过所有测试。

每次看到重复代码，都代表遗漏了抽象。重复的代码可能成为子程序或干脆是另一个类。将重复代码叠放进类似的抽象，增加了你的设计语言的词汇量。其他程序员可以用到你创建的抽象设施。编码变得越来越快，错误越来越少，因为你提升了抽象层级。

重复最明显的形态是你不断看到明显一样的代码，就像是某位程序员疯狂地用鼠标不断复制粘贴

代码。可以用单一方法来替代之。

较隐蔽的形态是在不同模块中不断重复出现、检测同一组条件的switch/case或if/else链。可以用多态来替代之。

更隐蔽的形态是采用类似算法但具体代码行不同的模块。这也是一种重复，可以使用模板方法模式 [4] 或策略模式 [5] 来修正。

的确，过去15年内出现的多数设计模式都是消除重复的有名手段。考德范式（Codd Normal Forms）是消除数据库规划中的重复的策略。OO自身也是组织模块和消除重复的策略。毫不出奇，结构化编程也是。

重点已经在那里了。尽可能找到并消除重复。

G6：在错误的抽象层级上的代码

创建分离较高层级一般性概念与较低层级细节概念的抽象模型，这很重要。有时，我们创建抽象类来容纳较高层级概念，创建派生类来容纳较低层次概念。这样做的时候，需要确保分离完整。所有较低层级概念放在派生类中，所有 较高层级概念放在基类中。

例如，只与细节实现有关的常量、变量或工具函数不应该在基类中出现。基类应该对这些东西一无所知。

这条规则对于源文件、组件和模块也适用。良好的软件设计要求分离位于不同层级的概念，将它们放到不同容器中。有时，这些容器是基类或派生类，有时是源文件、模块或组件。无论哪种情况，分离都要完整。较低层级概念和较高层级概念不应混杂在一起。看看下面的代码：

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

函数percentFull位于错误的抽象层级。尽管存在许多在其中“充满”（fullness）概念有意义的Stack的实现，但也有其他不能知道 自己有多满的实现存在。所以，该函数最好是放在类似BoundedStack之类的派生接口中。

你或许会认为，如果堆栈无边界，实现可以返回0。问题是，不存在真的无边界的堆栈。你不能真的避免在做以下检查时出现

OutOfMemoryException异常：

```
stack.percentFull() < 50.0.
```

实现返回0的函数可能是在撒谎。

要点是你不能就错误放置的抽象模型撒谎。孤立抽象是软件开发者最难做到的事之一，而且一旦做错也没有快捷的修复手段。

G7：基类依赖于派生类

将概念分解到基类和派生类的最普遍的原因是较高层级基类概念可以不依赖于较低层级派生类概念。这样，如果看到基类提到派生类名称，就可能发现了问题。通常来说，基类对派生类应该一无所知。

当然也有例外。有时，派生类数量严格固定，而基类中拥有在派生类之间选择的代码。在有限状态机的实现中这种情形很多见。然而，在那种情况下，派生类和基类紧密耦合，总是在同一个jar文件中部署。一般情况下，我们会想要把派生类和基类部署到不同的jar文件中。

将派生类和基类部署到不同的jar文件中，确保

基类jar文件对派生类jar文件的内容一无所知，我们就能把系统部署为分散和独立的组件。修改了这些组件时，不必重新部署基组件就能部署它们。这意味着修改产生的影响极大地降低了，而维护系统也变得更加简单。

G8： 信息过多

设计良好的模块有着非常小的接口，让你能事半功倍。设计低劣的模块有着广阔、深入的接口，你不得不事倍功半。设计良好的接口并不提供许多需要依靠的函数，所以耦合度也较低。设计低劣的借口提供大量你必须调用的函数，耦合度较高。

优秀的软件开发人员学会限制类或模块中暴露的接口数量。类中的方法越少越好。函数知道的变量越少越好。类拥有的实体变量越少越好。

隐藏你的数据。隐藏你的工具函数。隐藏你的常量和你的临时变量。不要创建拥有大量方法或大量实体变量的类。不要为子类创建大量受保护变量和函数。尽力保持接口紧凑。通过限制信息来控制耦合度。

G9： 死代码

死代码就是不执行的代码。可以在检查不会发

生的条件的if语句体中找到。可以在从不抛出异常的try语句的catch块中找到。可以在从不被调用的小工具方法中找到，也可以在永不会发生的switch/case条件中找到。

死代码的问题是过不久它就会发出臭味。时间越久，味道就越酸臭。这是因为，在设计改变时，死代码不会随之更新。它还能通过编译，但并不会遵循较新的约定或规则。它编写的时候，系统是另一番模样。如果你找到死代码，就体面地埋葬它，将它从系统中删除掉。

G10：垂直分隔

变量和函数应该在靠近被使用的地方定义。本地变量应该正好在其首次被使用的位置上面声明，垂直距离要短。本地变量不该在其被使用之处几百行以外声明。

私有函数应该刚好在其首次被使用的位置下面定义。私有函数属于整个类，但我们还是要限制调用和定义之间的垂直距离。找个私有函数，应该只是从其首次被使用处往下看一点那么简单。

G11：前后不一致

从一而终。这可以追溯到最小惊异原则。小心

选择约定，一旦选中，就小心持续遵循。

如果在特定函数中用名为response的变量来持有HttpServletResponse对象，则在其他用到HttpServletResponse对象的函数中也用同样的变量名。如果将某个方法命名为processVerificationRequest，则给处理其他请求类型的方法取类似的名字，例如processDeletionRequest。

如此简单的前后一致，一旦坚决贯彻，就能让代码更加易于阅读和修改。

G12：混淆视听

没有实现的默认构造器有何用处呢？它只会用无意义的杂碎搞乱对代码的理解。没有用到的变量，从不调用的函数，没有信息量的注释，等等，这些都是应该移除的废物。保持源文件整洁，良好地组织，不被搞乱。

G13：人为耦合

不互相依赖的东西不该耦合。例如，普通的enum不应在特殊类中包括，因为这样一来应用程序就要了解这些更为特殊的类。对于在特殊类中声明一般目的的static函数也是如此。

一般来说，人为耦合是指两个没有直接目的之间的模块的耦合。其根源是将变量、常量或函数不恰当地放在临时方便的位置。这是种漫不经心的偷懒行为。

花点时间研究应该在什么地方声明函数、常量和变量。不要为了方便随手放置，然后置之不理。

G14：特性依恋

这是Martin Fowler提出的代码味道之一 [6] 。类的方法只应对其所属类中的变量和函数感兴趣，不该垂青其他类中的变量和函数。当方法通过某个其他对象的访问器和修改器来操作该对象内部数据，则它就依恋于 该对象所属类的范围。它期望自己在那个类里面，这样就能直接访问它操作的变量。例如：

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

方法calculateWeeklyPay伸手到HourlyEmployee对象，获取要操作的数据。方法calculateWeeklyPay依恋于 HourlyEmployee的作用范围。它“期望”自己在HourlyEmployee中。

同样情况下，我们要消除特性依恋，因为它将一个类的内部情形暴露给了另外一个类。不过，有时特性依恋是种有必要的恶行。看下面的代码：

```
public class HourlyEmployeeReport {
  private HourlyEmployee employee ;

  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
        "Name: %s\tHours:%d.%1d\n",
        employee.getName(),
        employee.getTenthsWorked()/10,
        employee.getTenthsWorked()%10);
  }
}
```

显然，reportHours方法依恋于HourlyEmployee类。另一方面，我们并不想要HourlyEmployee得知报告的格式。把格式化字符串移到HourlyEmployee会破坏好几种面向对象设计原则 [7] 。它将把HourlyEmployee与报告的格式耦合起来，向该格式的修改暴露这个类。

G15：选择算子参数

没有什么比在函数调用末尾遇到一个false参数更为可憎的事情了。那个false是什么意思？如果它是true，会有什么变化吗？不仅是一个选择算子（selector）参数的目的难以记住，每个选择算子参数将多个函数绑到了一起。选择算子参数只是一种避免把大函数切分为多个小函数的偷懒做法。考虑下面这段代码：

```
public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

当加班时间以一倍半计算薪资时，用true调用这个函数，false则表示直接计算。每次用到这个函数，你都得记住calculateWeeklyPay(false)表示什么，这已经足够糟糕了。但这种函数真正的坏处在于作者错过了这样写的机会：

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}
```

```
public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

当然，选择算子不一定是boolean类型。可能是枚举元素、整数或任何一种用于选择函数行为的参数。使用多个函数，通常优于向单个函数传递某些代码来选择函数行为。

G16：晦涩的意图

代码要尽可能具有表达力。联排表达式、匈牙利语标记法和魔术数都遮蔽了作者的意图。例如，下面是overTimePay函数可能的一种表现形式：

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
  (int) Math.round(0.5 * iThsRte *
     Math.max(0, iThsWkd - 400)
  );
}
```

它既短小又紧凑，但实际上不可捉摸。值得花

时间将代码的意图呈现给读者。

G17：位置错误的权责

软件开发者做出的最重要决定之一就是在哪里放代码。例如，PI常量放在何处？是该在Math类中吗？或者应该属于Trigonometry类？还是在Circle类？

最小惊异原则在这里起作用了。代码应该放在读者自然而然期待它所在的地方。PI常量应该在出现在声明三角函数的地方。OVERTIME_RATE常量应该在HourlyPayCalculator类中声明。

有时，我们“聪明”地知道在何处放置功能代码。我们会放在自己方便而读者不能随直觉找到的地方。例如，也许我们需要打印出某个雇员的总工作时间的报表。我们可以在打印报表的代码中做工作时间统计，或者我们可以在接受工作时间卡的代码中保留一份工作时间记录。

做这个决定的途径之一是看函数名称。比如，报表模块有个名为getTotalHours的函数。接受时间卡的模块有一个saveTimeCard函数。顾名思义，哪个名称暗示了函数会计算总时间呢？答案显而易见。

显然，对于总时间应该在接受时间卡的时候计算而不是在打印报表时计算，这里面有些性能上的考量。没问题，但函数名称应该反映这种考虑。例如，应该在时间卡模块中有个computeRunningTotalOfHours函数。

G18：不恰当的静态方法

Math.max(double a, double)是个良好的静态方法。它并不在单个实体上操作；的确，不得不写new Math( ).max(a,b)甚至a.max(b)实在愚蠢。那个max用到的全部数据来自其两个参数，而不是来自“所属”对象。而且，我们也没机会 用到Math.max的多态特征。

不过，我们有时也编写不该是静态的静态方法。例如：

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

这看起来像是个有道理的static函数。它并不在任何特定对象上操作，而且从参数中获得全部数据。然而，我们却有理由希望这个函数是多态的。我们可能希望为计算每小时支付工资实现几种不同算法，例如OvertimeHourlyPayCalculator和

StraightTimeHourlyPayCalculator。所以，在这种情况下，该函数就不该是静态的。它该是Employee的非静态成员函数。

通常应该倾向于选用非静态方法。如果有疑问，就是用非静态函数。如果的确需要静态函数，确保没机会打算让它有多态行为。

G19：使用解释性变量

Kent Beck在其巨著*Smalltalk Best Practice Patterns* [8] 和另一部巨著*Implementation Patterns*（中译版《实现模式》）[9] 中都写到这个。让程序可读的最有力方法之一就是将计算过程打散成在用有意义的单词命名的变量中放置的中间值。

看看来自FitNesse的这个例子：

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

解释性变量的这种简单用法，说明了第一个匹配组是key，而第二个匹配组是value。

这事很难做过火。解释性变量多比少好。只要把计算过程打散成一系列良好命名的中间值，不透明的模块就会突然变得透明，这很值得注意。

G20：函数名称应该表达其行为

看看这行代码：

```
Date newDate = date.add(5);
```

你会期望它向日期添加5天吗？或者是5个星期？5个小时？该date实体会变化吗？或者该函数只是返回一个新的Date实体，并不改动旧的？从函数调用中看不出函数的行为。

如果函数向日期添加5天并且修改该日期，就该命名为addDaysTo或increaseByDays。如果函数返回一个表示5天后的日期，而不修改日期实体，就该叫做daysLater或daysSince。

如果你必须查看函数的实现（或文档）才知道它是做什么的，就该换个更好的函数名，或者重新安排功能代码，放到有较好名称的函数中。

G21：理解算法

好多可笑代码的出现，是因为人们没花时间去理解算法。他们硬塞进足够多的if语句和标识，从不真正停下来考虑发生了什么，勉强让系统能工作。

编程常常是一种探险。你以为 自己知道某事的正确算法，然后就卷起袖子瞎干一气，搞到“可以工作”为止。你怎么知道它“可以工作”？因为它通过了你能想到的单元测试。这种做法没错。实际上，这也是让函数按你设想的方式执行的唯一途径。不过，“可以工作”周围的引号可不能一直保留。

在你认为自己完成某个函数之前，确认自己理解了 它是怎么工作的。通过全部测试还不够好。你必须知道 [10] 解决方案是正确的。

获得这种知识和理解的最好途径，往往是重构函数，得到某种整洁而足具表达力、清楚呈示如何工作的东西。

G22：把逻辑依赖改为物理依赖

如果某个模块依赖于另一个模块，依赖就该是物理上的而不是逻辑上的。依赖者模块不应对被依赖者模块有假定（换言之，逻辑依赖）。它应当明

确地询问后者全部信息。

例如，想像你在编写一个打印出雇员工作时长的纯文本报表的函数。有个名为HourlyReporter的类把数据收集为某种方便的形式，传递到HourlyReportFormatter中，再打印出来。（如代码清单17-1所示。）

代码清单17-1　HourlyReporter.java

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
  }

  private void printAndClearItemList() {
    formatter.format(page);
    page.clear();
  }

  private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
```

```
    item.tenths = e.getTenthsWorked() % 10;
    page.add(item);
  }

  public class LineItem {
    public String name;
    public int hours;
    public int tenths;
  }
}
```

这段代码有尚未物理化的逻辑依赖。你能指出来吗？那就是常量PAGE_SIZE。HourlyReporter为什么要知道页面尺寸？页面尺寸只该是HourlyReportFormatter的权责。

PAGE_SIZE在HourlyReporter中声明，代表了一种位置错误的权责[G17]，导致HourlyReporter假定它知道页面尺寸。这类假设是一种逻辑依赖。HourlyReporter依赖于HourlyReportFormatter能应付55的页面尺寸。如果HourlyReportFormatter的某些实现不能处理这样的尺寸，就会出错。

可以通过创建HourlyReport中名为getMaxPageSize( )的新方法来物理化这种依赖。HourlyReporter将调用这个方法，而不是使用PAGE_SIZE常量。

G23：用多态替代If/Else或Switch/Case

有了第6章谈及的主题，这条建议看似奇怪。在那章中，我提出在添加新函数甚于添加新类型的系统中，switch语句是恰当的。

首先，多数人使用switch语句，因为它是最直截了当又有力的方案，而不是因为它适合当前情形。这给我们的启发是在使用switch之前，先考虑使用多态。

其次，函数变化甚于类型变化的情形相对罕见。每个 switch语句都值得怀疑。

我使用所谓“单个switch”规则：对于给定的选择类型，不应有多于一个**switch**语句。在那个**switch**语句中的多个**case**，必须创建多态对象，取代系统中其他类似**switch**语句。

G24：遵循标准约定

每个团队都应遵循基于通用行业规范的一套编码标准。编码标准应指定诸如在何处声明实体变量，如何命名类，方法和变量，在何处放置括号，等等。团队不应用文档描述这些约定，因为代码本身提供了范例。

团队中的每个成员都应遵循这些约定。这意味着每个团队成员必须成熟到能了解只要全体同意在

何处放置括号，那么在哪里放置都无关紧要。

如果你想知道我遵循哪些约定，可以查看代码清单B-7～B-14中重构之后的代码。

G25：用命名常量替代魔术数

这大概是软件开发中最古老的规则之一了。我记得，在20世纪60年代介绍COBOL、FORTRAN和PL/1的手册中就读到过。在代码中出现原始形态数字通常来说是坏现象。应该用良好命名的常量来隐藏它。

例如，数字86400应当藏在常量SECONDS_PER_DAY后面。如果每页打印55行，则常数55应该藏在常量LINES_PER_PAGE后面。

有些常量与非常具有自我解释能力的代码协同工作时，如此易于识别，也就不必总是需要命名常量来隐藏了。例如：

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

在上例中，我们真需要常量

FEET_PER_MILE、WORK_HOURS_PER_DAY和TWO吗？显然，最后那个很可笑。有些情况下，常量直接写作原始形态数字会更好。你可能会质疑WORK_HOURS_PER_ DAY，因为约定规则可能会改变。另一方面，在这里直接用数字8读起来很舒服，也就没必要非用17个额外的字母来加重读者负担不可。对于FEET_PER_MILE，数字5280众人皆知，意义独特，即便没有上下文环境，读者也能识别它。

3.141592653589793之类常数也众所周知，很容易识别。不过，如果直接使用原始形式，却很有可能出错。每次有人看到3.141592653589793，都会知道那是π值，从而不会去仔细查看。（你发现那个错误的数字了吗？）我们不想要人们使用3.14、3.14159或3.142等。所以，为我们定义好Math.PI是件好事。

术语“魔术数”不仅是说数字。它泛指任何不能自我描述的符号。例如：

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

上列断言中有两个魔术数。第一个显然是777，它的意义并不明确。第二个魔术数是John

Doe，因为其意图不明显。

John Doe是开发团队创建的测试数据中编号为#7777的雇员。团队中每个成员都知道，当连接到数据库时，里面已经有数个雇员信息，其值和属性都是大家熟知的。所以，这个测试应该读作：

```
assertEquals(
  HOURLY_EMPLOYEE_ID,
  Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26：准确

期望某个查询的第一次匹配就是唯一 匹配可能过于天真。用浮点数表示货币几近于犯罪。因为你不想做并发更新就避免使用锁和/或事务管理往好处说也是一种懒惰行为。在可以用List的时候非要把变量声明为ArrayList就过分拘束了。把所有变量设置为protected却不够自律。

在代码中做决定时，确认自己足够准确 。明确自己为何要这么做，如果遇到异常情况如何处理。别懒得理会决定的准确性。如果你打算调用可能返回null的函数，确认自己检查了null值。如果查询你认为是数据库中唯一的记录，确保代码检查不存在其他记录。如果要处理货币数据，使用整数

[11]，并恰当地处理四舍五入。如果可能有并发更新，确认你实现了某种锁定机制。

代码中的含糊和不准确要么是意见不同的结果，要么源于懒惰。无论原因是什么，都要消除。

G27：结构甚于约定

坚守结构甚于约定的设计决策。命名约定很好，但却次于强制性的结构。例如，用到良好命名的枚举的switch/case要弱于拥有抽象方法的基类。没人会被强迫每次都以同样方式实现switch/case语句，但基类却让具体类必须实现所有抽象方法。

G28：封装条件

如果没有if或while语句的上下文，布尔逻辑就难以理解。应该把解释了条件意图的函数抽离出来。

例如：

```
if (shouldBeDeleted(timer))
```

要好于

```
if  (timer.hasExpired()  &&  !timer.isRecurrent())
```

G29：避免否定性条件

否定式要比肯定式难明白一些。所以，尽可能将条件表示为肯定形式。例如：

```
if  (buffer.shouldCompact())
```

要好于

```
if  (!buffer.shouldNotCompact())
```

G30：函数只该做一件事

编写执行一系列操作的包括多段代码的函数常常是诱人的。这类函数做了不只一件事，应该转换为多个更小的函数，每个只做一件事。

例如：

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
```

```
    }
  }
}
```

这段代码做了三件事。它遍历所有雇员，检查是否该给雇员付工资，然后支付薪水。代码可以写得更好，如：

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

上列每个函数都只做一件事。（见前文“只做一件事”一节。）

G31：掩蔽时序耦合

常常有必要使用时序耦合，但你不应该掩蔽它。排列函数参数，好让它们被调用的次序显而易见。看下列代码：

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
  }
  ...
}
```

三个函数的次序很重要。捕鱼之前先织网，织网之前先编绳。不幸的是，代码并没有强制这种时序耦合。其他程序员可以在调用saturateGradient之前调用reticulateSplines，从而导致抛出UnsaturatedGradientException异常。更好的方式是:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  ...
}
```

这样就通过创建顺序队列暴露了时序耦合。每

个函数都产生出下一个函数所需的结果，这样一来就没理由不按顺序调用了。

你可能会抱怨着增加了函数的复杂度，没错，不过这点额外的复杂度却曝露了该种情况真正的时序复杂性。

注意我保留了那些实体变量。我假设类中的私有方法可能会用到它们。即便如此，我还是希望参数能让时序耦合变得可见。

G32：别随意

构建代码需要理由，而且理由应与代码结构相契合。如果结构显得太随意，其他人就会想修改它。如果结构自始至终保持一致，其他人就会使用它，并且遵循其约定。例如，我最近对FitNesse做合并修改，发现有位贡献者这么做：

```
public class AliasLinkWidget extends ParentWidget
{
  public static class VariableExpandingWidgetRoot {
    ...

  ...
}
```

问题在于，VariableExpandingWidgetRoot没必

要在AliasLinkWidget作用范围之内。而且，其他无关的类也用到AliasLinkWidget.VariableExpandingWidgetRoot。这些类没必要了解AliasLinkWidget。

或许那位程序员只是循例把VariableExpandingWidgetRoot放到AliasWidget里面，或者他真认为这么做是对的。不管原因是什么，结果都显得随心所欲。不作为类工具的公共类，不应该放到其他类里面。惯例是将它置为public，并且放在代码包的顶部。

G33：封装边界条件

边界条件难以追踪。把处理边界条件的代码集中到一处，不要散落于代码中。我们不想见到四处散见的+1和−1字样。看看这个来自FIT的简单例子：

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

注意，level + 1出现了两次。这是个应该封装到名为nextLevel之类的变量中的边界条件。

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34： 函数应该只在一个抽象层级上

函数中的语句应该在同一抽象层级上，该层级应该是函数名所示操作的下一层。这可能是最难理解和遵循的启发。尽管概念足够直白，人们还是很容易混淆抽象层级。例如，请看下面来自FitNesse的例子：

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");

  return html.toString();
}
```

稍微研究一下，你就会看到发生了什么。该函数构建了绘制横贯页面线条的HTML标记。线条高度在size变量中指定。

再看一遍。方法混杂了至少两个抽象层级。第

一个是横线有尺寸这个概念。第二个是hr标记自身的语法。这段代码来自FitNesse的HruleWidget模块。该模块检测一行4个或更多个破折号，并将其转换为恰当的hr标记。破折号越多，尺寸越大。

我重构了这段代码。注意，我修改了size字段的名称，反映其真正目的。它表示额外破折号的数量。

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

这次修改很好地拆开了两个抽象层级。函数render只构造一个hr标记，不去管该标记的HTML语法。而HtmlTag模块则照管所有这些肮脏的语法问题。

做出修改时，我发现了一处微小的错误。原始代码没有加上hr标记的结束斜线符，而XHTML标

准要求这样做。（换言之，代码使用了<hr>而不是<hr />。）HtmlTag模块很早就改造成符合XHTML标准了。

拆分不同抽象层级是重构的最重要功能之一，也是最难做的一个。以下面的代码为例。这是我第一次尝试拆分HruleWidget.rendermethod中的抽象层级的结果。

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

此时，我的目的是做必要的拆分，并让测试通过。我轻易达到了这一目的，但结果是该函数仍然混杂了多个抽象层级。此时，混杂的层级是hr标记的构建，以及size变量的翻译和格式化。这说明当你循抽象界线拆解函数时，经常会挖出原本被之前的结构所掩蔽的新抽象界线。

G35：在较高层级放置可配置数据

如果你有个已知并该在较高抽象层级的默认常

量或配置值，不要将它埋藏到较低层级的函数中。把它作为较高层级函数调用较低层级函数时的一个参数。看看以下来自FItNesse的代码：

```
  public static void main(String[] args) throws Exception
  {
    Arguments arguments = parseCommandLine(args);
    ...
  }

public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "FitNesseRoot";
  public static final int DEFAULT_PORT = 80;
  public static final int DEFAULT_VERSION_DAYS = 14;
  ...
}
```

命令行参数在FitNesse中的第一行可执行代码得到解析。这些参数的默认值在Argument类的顶部指定。你不必到系统的较低层级去查看类似的语句：

```
if (arguments.port == 0) // use 80 by default
```

位于较高层级的配置性常量易于修改。它们向下贯穿应用程序。应用程序的较低层级并不拥有这些常量的值。

G36：避免传递浏览

通常我们不想让某个模块了解太多其协作者的信息。更具体地说，如果A与B协作，B与C协作，我们不想让使用A的模块了解C的信息。（例如，我们不想写类似a.getB( ).getC( ).doSomething( )的代码。）

这就是所谓得墨忒耳律。The Pragmatic Programmers（中译版《程序员修炼之道》）称之为“编写害羞代码” [12] 。两者都归结为确保模块只了解其直接协作者，不了解整个系统的游览图。

如果有多个模块使用类似a.getB( ).getC( )这样的语句形式，就难以修改设计和架构，在B和C之间插进一个Q。你得找到a.getB( ).getC( )出现的所有地方，并将其改为a.getB( ).getQ( ).getC( )。系统就此变得缺乏柔韧性。太多的模块了解了太多有关架构的信息。

正确的做法是让直接协作者提供所需的全部服务。不必逛遍系统的对象全图，搜寻我们要调用的方法。只要简单地说：

```
myCollaborator.doSomething().
```

17.5　Java

J1：通过使用通配符避免过长的导入清单

如果使用了来自同一程序包的两个或多个类，用以下语句导入整个包：

```
import package.*;
```

过长的导入清单令读者望而却步。我们不想用80行导入语句搞乱模块顶部位置。我们想要导入语句简约地列出我们要使用的包。

指定导入包是种硬依赖，而通配符导入则不是。如果你具体指定导入某个类，该类必须 存在。但如果你用通配符导入某个包，则不需要存在具体的类。导入语句只是在搜寻名称时把这个包列入查找路径。所以，这种导入并未构成真正的依赖，也就让我们的模块较少耦合。

有时，长长的具体导入清单也会有用。例如，如果你在处理遗留下来的代码，想要找出需要为哪些类构造替身类和占位代码，就可以遍历导入清单，找出这些类的真名，再恰当地放置占位代码。

不过，这种用法很罕见。而且，多数现代IDE允许你用一个命令就把通配符导入语句转换为指定导入清单。所以，即便在处理遗留代码时，最好也用通配符导入。

通配符导入有时会导致名称冲突和歧义。两个同名但位于不同包中的类需要指名导入，或至少在使用时指定名称。这种情形的确讨厌，不过很罕见，所以使用通配符导入通常仍优于指定名称导入。

J2：不要继承常量

我见过这种情况好几次，它总是让我面露苦笑。某个程序在接口中放了些常量， 再通过继承结构来访问这些常量。看看以下代码：

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
        hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

常量TENTHS_PER_WEEK和OVERTIME_RATE来自何方？它们可能来自Employee类。来看看：

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

不，不在那儿。不过在哪儿呢？再仔细看Employee类。它实现了PayrollConstants接口。

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

真是丑陋不堪！常量躲在了继承结构的最顶端。呸！别利用继承欺骗编程语言的作用范围规则。应该用静态导入。

```
import static PayrollConstants.*;



public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;
```

```
  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3：常量 vs. 枚举

现在enum已经加入Java语言（Java 5），放心用吧！别再用那个public static final int老花招。那样做int的意义就丧失了，而用enum则不然，因为它们隶属于有名称的枚举。

而且，仔细研究enum的语法。它可以拥有方法和字段，从而成为能比int提供更多表达力和灵活性的强有力工具。看看以下发薪代码中的不同做法：

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
 HourlyPayGrade grade;




  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
        grade.rate
```

```
() * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}

public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  LEUTENANT_JOURNEYMAN {
    public double rate() {
      return 1.2;
    }
  },
  JOURNEYMAN {
    public double rate() {
      return 1.5;
    }
  },
  MASTER {
    public double rate() {
      return 2.0;
    }
  };

  public abstract double rate();
}
```

17.6　名称

N1：采用描述性名称

不要太快取名。确认名称具有描述性。记住，事物的意义随着软件的演化而变化，所以，要经常性地重新估量名称是否恰当。

这不仅是一条“感觉良好式”建议。软件中的名称对于软件可读性有90%的作用。你要花时间明智地取名，保持名称有关。名称太重要了，不可随意对待。

看看以下代码。这段代码是做什么的？用了好名称的代码一目了然，而这样的代码却是符号和魔术数的大杂烩。

```
public int x() {
  int q = 0;
  int z = 0;
  for (int kk = 0; kk < 10; kk++) {
    if (l[z] == 10)
    {
      q += 10 + (l[z + 1] + l[z + 2]);
      z += 1;
    }
    else if (l[z] + l[z + 1] == 10)
    {
      q += 10 + l[z + 2];
      z += 2;
```

```
    } else {
      q += l[z] + l[z + 1];
      z += 2;
    }
  }
  return q;
}
```

下面是这段代码应该写成的样子。代码片段实际上不如上段完整。但你还是能马上推断出它要做什么，而且很有可能依据推断出的意思写出遗漏的函数。魔术数不复神秘，算法的结构也足具描述性。

```
public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
      frame += 2;
    }
  }
  return score;
}
```

仔细取好的名称的威力在于，它用描述性信息

覆盖了代码。这种信息覆盖设定了读者对于模块中其他函数行为的期待。看看上面的代码，你就能推断出isStrike( )的实现。读到isStrick方法时，它“深合你意”[13] 。

```
private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

N2：名称应与抽象层级相符

不要取沟通实现的名称；取反映类或函数抽象层级的名称。这样做不容易。人们擅长于混杂抽象层级。每次浏览代码，你总会发现有些变量的名称层级太低。你应当趁机为之改名。要让代码可读，需要持续不断的改进。看看下面的Modem接口：

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

粗看还行。函数看来都很合适，对于多数应用程序来说是这样。不过，想想看某个应用中有些调制解调器并不用拨号连接的情形。有些用线缆直连

（就像如今为多数家庭提供Internet连接的线缆解调器）的情形。有些通过向USB口发送端口信息连接。显然，有关电话号码的信息就是位于错误的抽象层级了。对于这种情形，更好的命名策略可能是：

```
public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

现在名称再不与电话号码有关系。还是可以用于用电话号码的情形，也可以用于其他连接策略。

N3：尽可能使用标准命名法

如果名称基于既存约定或用法，就比较易于理解。例如，如果你采用油漆工模式，就该在给油漆类命名时用上Decorator字样。例如，AutoHangupModemDecorator可能是某个给Modem类刷上在会话结束时自动挂机的能力的类的名称。

模式只是标准的一种。例如，在Java中，将对象转换为字符串的函数通常命名为toString。最好是遵循这些约定，而不是自己创造命名法。

对于特定项目，开发团队常常发明自己的命名标准系统。Eric Evans称之为项目的共同语言[14]。代码应该使用来自这种语言的术语。简言之，具有与项目有关的特定意义的名称用得越多，读者就越容易明白你的代码是做什么的。

N4：无歧义的名称

选用不会混淆函数或变量意义的名称。看看来自FitNesse的这个例子：

```
private String doRename() throws Exception
{
  if(refactorReferences)
    renameReferences();
  renamePage();

  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

该函数的名称含混不清，没有说明函数的作用。由于在doRename函数里面还有个名为renamePage的函数，这就更不明白了！这些名称有没有说明两个函数之间的区别呢？没有。

该函数的更好名称应该是renamePageAndOptionallyAllReferences。看似太

长，的确也很长，不过它只在模块中的一处被调用，所以其解释性的好处大过了长度的坏处。

N5：为较大作用范围选用较长名称

名称的长度应与作用范围的广泛度相关。对于较小的作用范围，可以用很短的名称，而对于较大作用范围就该用较长的名称。

类似i和j之类的变量名对于作用范围在5行之内的情形没问题。看看以下来自老“标准保龄球游戏”的代码片段：

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
    g.roll(pins);
}
```

这段代码很明白，如果用rollCount之类烦人的名称代替变量i，反而是徒增混乱。另一方面，在较长距离上，使用短名称的变量和函数会丧失其含义。名称的作用范围越大，名称就该越长、越准确。

N6：避免编码

不应在名称中包括类型或作用范围信息。在如今的开发环境中，m_或f之类前缀完全无用。类似vis_（表示图形系统）之类的项目或子系统名称也属多余。当今的开发环境不用纠缠于名称也能提供这些信息。不要用匈牙利语命名法污染你的名称。

N7：名称应该说明副作用

名称应该说明函数、变量或类的一切信息。不要用名称掩蔽副作用。不要用简单的动词来描述做了不止一个简单动作的函数。例如，请看以下来自TestNG的代码：

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

该函数不只是获取一个oos，如果oos不存在，还会创建一个。所以，更好的名称大概是createOrReturnOos。

17.7　测试

T1：测试不足

　　一套测试中应该有多少个测试？不幸的是，许多程序员的衡量标准是“看起来够了”。一套测试应该测到所有可能失败的东西。只要还有没被测试探测过的条件，或是还有没被验证过的计算，测试就还不够。

T2：使用覆盖率工具

　　覆盖率工具能汇报你测试策略中的缺口。使用覆盖率工具能更容易地找到测试不足的模块、类和函数。多数IDE都给出直观的指示，用绿色标记测试覆盖了的代码行，而未覆盖的代码行则是红色。这样就能又快又容易地找到尚未检测过的if或catch语句。

T3：别略过小测试

　　小测试易于编写，其文档上的价值高于编写成本。

T4：被忽略的测试就是对不确定事物的疑问

有时，我们会因为需求不明而不能确定某个行为细节。可以用注释掉的测试或者用@Ignore标记的测试来表达我们对于需求的疑问。使用哪种方式，取决于该不确定性所关涉代码是否要编译。

T5：测试边界条件

特别注意测试边界条件。算法的中间部分正确但边界判断错误的情形很常见。

T6：全面测试相近的缺陷

缺陷趋向于扎堆。在某个函数中发现一个缺陷时，最好全面测试那个函数。你可能会发现缺陷不止一个。

T7：测试失败的模式有启发性

有时，你可以通过找到测试用例失败的模式来诊断问题所在。这也是尽可能编写足够完整的测试用例的理由之一。完整的测试用例，按合理的顺序排列，能暴露出模式。

简单举例，假设你注意到所有长于5个字符的输入都会导致测试失败，或者向函数的第二个参数传入负数都会导致测试失败。有时，只要看看测试报告的红绿模式，就足以绽放出那句带来解决方法

的“啊哈！”回头看看第16章“重构SerialDate”中的有趣例子吧。

T8：测试覆盖率的模式有启发性

查看被或未被已通过的测试执行的代码，往往能发现失败的测试为何失败的线索。

T9：测试应该快速

慢速的测试是不会被运行的测试。时间一紧，较慢的测试就会被摘掉。所以，竭尽所能 让测试够快。

17.8　小结

这份启发与味道的清单很难说已完备无缺。我不能确定这样一份清单会不会 完备无缺。但或许完整性不该是目标，因为该清单确实 给出了一套价值体系。

那套价值体系才该是目标，也是本书的主题所在。整洁代码并非遵循一套规则写就。学习一系列启发并不足以让你成为软件匠人。专业性和技艺来自于驱动规程的价值观。

17.9　文献

[Refactoring]：*Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

[PRAG]：*The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[GOF]：*Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[Beck97]：*Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[Beck07]：*Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.

[PPP]：*Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[DDD]：*Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

---

[1]　原注：[Refactoring]。

[2]　原注：或称“最少惊诧原则”（The Principle of Least Astonishment）：http://en.wikipedia.org/wiki/Principle_of_least_a
。

[3]　原注：[FRAG]。

[4]　原注：[GOF]。

[5]　原注：[GOF]。

[6]　原注：[Refactoring]。

[7]　原注：具体是单一权责原则，开放闭合原则和公共关闭原则。参见[PPP]。

[8]　原注：[Beck97]，p. 108。

[9]　原注：[Beck07]。

[10]　原注：了解代码如何工作与了解算法是否按需要执行是不一样的。不确定算法是否恰当司空见惯，而不确定代码做什么却是一种懒惰行为。

[11]　原注：或者用更好的使用整数的Money类。

[12]　原注：[PRAG]，P. 138。

[13]　原注：见第一章中Ward Cunningham的引语。

[14]　原注：[DDD]。

附录**A**　并发编程**II**

Brett L.Schuchert

本附录扩充了“并发编程”一章的内容，由一组相互独立的主题组成，你可以按随意顺序阅读。为了实现这样的阅读方式，节与节之间存在一些重复内容。

A.1 客户端/服务器的例子

想像一个简单的客户端/服务器应用程序。服务器在一个套接字上等待接受来自客户端的连接请求。客户端连接到服务器并发送请求。

A.1.1 服务器

下面是服务器应用程序的简化版本代码。在后文“客户端/服务器非多线程版本”一节中有完整的代码。

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
  try {
    Socket socket = serverSocket.accept();
    process(socket);
  } catch (Exception e) {
    handle(e);
  }
}
```

这个简单的应用等待连接请求，处理接收到的新消息，再等待下一个客户端请求。下面是连接到服务器的客户端代码：

```
private void connectSendReceive(int i) {
```

```
    try {
      Socket socket = new Socket("localhost", PORT);
      MessageUtils.sendMessage(socket, Integer.toString(i));
      MessageUtils.getMessage(socket);
      socket.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
}
```

这对客户端/服务器程序运行得如何呢？怎样才能正式地描述其性能？下面是断言其性能“可接受”的测试：

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
  Thread[] threads = createThreads();
  startAllThreadsw(threads);
  waitForAllThreadsToFinish(threads);
}
```

为了让例子够简单，设置过程被忽略了（见后文ClientText.java部分）。测试断言程序应该在10000毫秒内完成。

这是个验证系统吞吐量的典型例子。系统应该在10秒钟以内完成一组客户端请求。只要服务器能在时限内处理每个客户端请求，测试就通过了。

如果测试失败会怎样？缺少了某些事件轮询机

制，在单个线程上也没什么可让代码更快的手段。使用多线程能解决问题吗？可能会，我们先得了解什么地方耗费时间。下面是两种可能：

- I/O——使用套接字、连接到数据库、等待虚拟内存交换等；
- 处理器——数值计算、正则表达式处理、垃圾回收等。

以上在系统中都会部分存在，但对于特定的操作，其中之一会起主导作用。如果代码运行速度主要与处理器有关，增加处理器硬件就能提升吞吐量，从而通过测试。但CPU运算周期是有上限的，因此，只是增加线程的话并不会提升受处理器限制的代码的速度。

另一方面，如果吞吐量与I/O有关，则并发编程能提升运行效率。当系统的某个部分在等待I/O，另一部分就可以利用等待的时间处理其他事务，从而更有效地利用了CPU能力。

A.1.2 添加线程代码

假定性能测试失败了。如何才能提高吞吐量、通过性能测试呢？如果服务器的process方法与I/O有关，就有个办法让服务器利用线程（只需要修改

processMessage）：

```
void process(final Socket socket) {
  if (socket == null)
    return;

  Runnable clientHandler = new Runnable() {
    public void run() {
      try {
        String message = MessageUtils.getMessage(socket);
        MessageUtils.sendMessage(socket, "Processed: " + message)
;
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  };

  Thread clientConnection = new Thread(clientHandler);
  clientConnection.start();
}
```

假设修改后测试通过了[1]。代码是否完整、正确了呢？

A.1.3 观察服务器端

修改了的服务器成功通过测试，只花费了一秒多钟时间。不幸的是，这种解决手段有点一厢情愿，而且导致了新问题产生。

服务器应该创建多少个线程？代码没有设置上

限，所以我们很有可能达到Java虚拟机（JVM）的限制。对于许多简单系统来说这无所谓。但如果系统要支持公众网络上的众多用户呢？如果有太多用户同时连接，系统就有可能挂掉。

不过先把性能问题放到一边吧。这种手段还有整洁性和结构上的问题。服务器代码有多少种权责呢？

- 套接字连接管理；
- 客户端处理；
- 线程策略；
- 服务器关闭策略。

这些权责不幸全在process函数中。而且，代码跨越多个抽象层级。所以，即便process函数这么短小，还是需要再加以切分。

服务器有几个修改的原因，所以它违反了单一权责原则。要保持并发系统整洁，应该将线程管理代码约束于少数几处控制良好的地方。而且，管理线程的代码只应该做管理线程的事。为什么？即便无需同时考虑其他非多线程代码，跟踪并发问题都已经足够困难了。

如果为上述每个权责（包括线程管理权责在

内）创建单独的类，当改动线程管理策略时，就会对整个代码产生较小影响，不至于污染其他权责。这样一来，也能在不担心线程问题的前提下测试所有其他权责。下面是修改过的版本：

```
public void run() {
  while (keepProcessing) {
    try {
      ClientConnection clientConnection = connectionManager.awaitClient();
      ClientRequestProcessor requestProcessor
      = new ClientRequestProcessor(clientConnection);
      clientScheduler.schedule(requestProcessor);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  connectionManager.shutdown();
}
```

所有与线程相关的东西都放到了clientScheduler里面。如果出现并发问题，只要看这个地方就好了：

```
public  interface  ClientScheduler  {
   void  schedule(ClientRequestProcessor  requestProcessor);
}
```

并发策略易于实现：

```
public class ThreadPerRequestScheduler implements ClientSchedul
er {
  public void schedule(final ClientRequestProcessor requestProce
ssor) {
    Runnable runnable = new Runnable() {
      public void run() {
        requestProcessor.process();
      }
    };

    Thread thread = new Thread(runnable);
    thread.start();
  }
}
```

把所有线程管理隔离到一个位置，修改控制线程的方式就容易多了。例如，移植到Java 5 Executor框架就只需要编写一个新类并插进来即可（如代码清单A-1所示）。

代码清单A-1　ExecutorClientScheduler.java

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler
 {
  Executor executor;

  public ExecutorClientScheduler(int availableThreads) {
    executor = Executors.newFixedThreadPool(availableThreads);
  }

  public void schedule(final ClientRequestProcessor requestProce
ssor) {
    Runnable runnable = new Runnable() {
      public void run() {
        requestProcessor.process();
```

```
      }
    };
    executor.execute(runnable);
  }
}
```

A.1.4　小结

本例介绍的并发编程，演示了一种提高系统吞吐量的方法，以及一种通过测试框架验证吞吐量的方法。将全部并发代码放到少数类中，是应用单一权责原则的范例。对于并发编程，因其复杂性，这一点尤其重要。

A.2　执行的可能路径

复查没有循环或条件分支的单行Java方法incrementValue：

```
public class IdGenerator {
  int lastIdUsed;

  public int incrementValue() {
    return ++lastIdUsed;
  }
}
```

忽略整数溢出的情形，假定只有单个线程能访问IdGenerator的单个实体。这种情况下，只有一种执行路径和一个确定的结果：

- 返回值等于lastIdUsed的值，两者都比调用方法前大1。

如果使用两个线程、不修改方法的话会发生什么？如果每个线程都调用一次incrementValue，可能得到什么结果呢？有多少种可能执行路径？首先来看结果（假定lastIdUsed初始值为93）：

- 线程1得到94，线程2得到95，lastIdUsed为95；

- 线程1得到95，线程2得到94，lastIdUsed为95；
- 线程1得到94，线程2得到94，lastIdUsed为94。

最后一个结果尽管令人吃惊，也是有可能出现的。要想明白为何可能出现这些结果，就需要理解可能执行路径的数量以及Java虚拟机是如何执行这些路径的。

A.2.1　路径数量

为了算出可能执行路径的数量，我们从生成的字节码开始研究。那行Java代码（return ++lastIdUsed;）变成了8个字节码指令。两个线程有可能交错执行这8个指令，就像庄家在洗牌时交错牌张一样[2]。即便每只手上只有8张牌，洗牌得到的结果数量也很可观。

对于指令系列中有N个指令和T个线程、没有循环或条件分支的简单情况，总的可能执行路径数量等于

$$\frac{(NT)!}{N!^T}$$

计算可能执行次序

以下摘自鲍勃大叔给Brett的一封电子邮件：

对于N步指令和T个线程，总共有T*N个步骤。在执行每步指令之前，会有在T个线程中选择其一的环境开关。因而每条路径都能以一个数字字符串的形式来表示该环境开关。对于步骤A、B及线程1和2，可能有6条可能路径：1122、1212、1221、2112、2121和2211。或者以指令步骤表示为A1B1A2B2、A1A2B1B2、A1A2B2B1、A2A1B1B2、A2A1B2B1及A2B2A1B1。对于三个线程，执行序列就是112233、112323、113223、113232、112233、121233、121323、121332、123132、123123……

这些字符串的特征之一是每个T总会出现N次。所以字符串111111是无效的，因为里面有6个1，而2和3则未出现过。

所以要排列组合N1、N2……直至NT。这其实就是N * T对应N*T的排列，即(N*T)!，但要剔除重复的情形。所以，巧妙之处就在于计算重复次数并从(N*T)!中剔除掉。

对于两步指令和两个线程，有多少重复呢？每个四位数字符串中都有两个1和两个2。每个这种配对都可以在不影响字符串意义的前提下调换。可以同时调换全部1和2，也可以都不调换。所以每个字符串就有四种同构形态，即存在3次重复。所以四分之三的路径是重复的；而四分之一的排列则不重复。4!*.25=6。这样计算看来可行。

> 有多少重复呢？对于N=1且T=2的情形，我可以调换1，调换2，或两者都调换。对于N=2且T=3的情形，我可以调换1、2、3，1和2，1和3，或2和3。调换只是N的排列组合罢了。设有N的P种排列组合。排列组合的方式总共有P**T种。
>
> 所以可能的同构形态数量为N!**T。路径的数量就是(T*N)!/(N!**T)。对于T=2且N=2的情况，结果就是6（即24/4）。
>
> 对于N=2且T=3，结果是720/8=90。
>
> 对于N=3且T=3，结果是9!/6^3=1680。

对于一行Java代码（等同于8行字节码）和两

个线程的简单情况，可能执行路径的总数量就是12870。如果lastIdUsed的类型为long，每次读/写操作都变成了两次操作，而可能的次序高达2704156种。

如果改动一下该方法会怎样？

```
public synchronized


 void incrementValue() {
    ++lastIdUsed;
}
```

这样一来，对于两个线程的情况，可能执行路径的数量就是2，即N!。

A.2.2　深入挖掘

两个线程都调用方法一次（在添加synchronize之前）、得到同一结果数字的惊异结果又怎样呢？怎么可能出现这种情况？一样一样来。

什么是原子操作？可以把原子操作定义为不可中断的操作。例如，在下列代码的第5行，0被赋值给lastid，就是一个原子操作。因为依据Java内存模型，32位值的赋值操作是不可中断的。

```
01:  public class Example {
02:    int lastId;
03:
04:    public void resetId()  {
05:      value = 0;
06:    }
07:
08:    public int getNextId()  {
09:      ++value;
10:    }
11:  }
```

如果把lastId的类型从int改为long会怎样？第5行还是原子操作吗？如果不考虑JVM规约，则有可能根据处理器不同而不同。不过，根据JVM规约，64位值的赋值需要两次32位赋值。这意味着在第一次和第二次32位赋值之间，其他线程可能插进来，修改其中一个值。

第9行的前递增操作符++又怎样呢？前递增操作符可以被中断，所以它不是原子的。为了理解这点，仔细复查一下这些方法的字节码吧。

在更进一步之前，有三个重要的定义：

- 框架——每个方法调用都需要一个框架。该框架包括返回地址、传入方法的参数，以及方法中定义的本地变量。这是定义一个调用堆栈的标准技术，现代编程语言用来实现基本函数/方

法调用和递归调用；

- 本地变量——方法作用范围内定义的每个变量。所有非静态方法至少有一个变量this，代表当前对象，即接收导致方法调用的（当前线程内）大多数最新消息的对象；
- 运算对象栈——Java虚拟机中的许多指令都有参数。运算对象栈是放置参数的地方。堆栈是个标准的后入先出（LIFO）数据结构。

下面是restId( )的字节码，如表A-1所示。

表**A-1　restId( )**的字节码

指令	描述	操作对象栈
ALOAD　0	将第0个变量放到操作对象栈中。什么是第0个变量？就是this，当前对象。当方法被调用，消息接收者，Example的一个实体，被推到为方法调用创建的框架的本地变量数组中。这总是放进每个实体方法的第一个变量	this
ICONST_0	将常量值0放到操作对象栈中	this,　0
PUTFIELD lastId	将堆栈中的第一个值（即0）存储到引用对象的字段值，距堆栈顶部this一个对象引用的距离	<empty>

这三个指令确保是原子的，因为尽管执行它们的线程可能在其中任何一个指令后被打断，但PUTFIELD指令（堆栈顶部的常量值0和顶端之下的this引用及其字段值）的信息并不能为其他线程所

触及。所以，当赋值操作发生时，值0一定将存储到字段值中。该操作是原子的。操作对象都处理对于方法而言是本地的信息，故在多个线程之间并无冲突。

所以，如果这三个指令由10个线程执行，就会有4.38679733629e+24种可能的执行次序。不过，只会有一种可能的结果，所以执行次序不同无关紧要。对于本例中的long常量，总是有同一种运算结果。为什么？因为10个线程的赋值操作都是针对一个常量的。即便它们互相干涉，结果也是一样。

方法getNextId中的++操作就会有问题了。假定lastId在方法开始时的值为42.下面是新方法的字节码，如表A-2所示。

表**A-2**　新方法的字节码

指令	描述	操作对象栈
ALOAD　0	将this装载到操作对象栈	this
DUP	复制堆栈顶部内容。在对象栈中有两个this的复本	this, this
GETFIELD lastId	从指向堆栈顶部（this）的对象中取得字段lastId的值，并存储回堆栈中	this, 42
ICONST_1	将整数常量1推入堆栈	this, 42, 1

IADD	对堆栈顶部的两个值做整数加操作，将结果存储回堆栈	this, 43
DUP_X1	复制值43，放到this之前	43, this, 43
PUTFIELD value	将堆栈顶部的值43放到当前对象的字段值中，表现为对象栈中的下一个值this	43
IRETURN	返回堆栈顶部（而且只是顶部）的值	<empty>

设想第一个线程完成了前三个操作，直到执行完GETFIELD，然后被打断。第二个线程接手并完成整个方法调用，lastId的值递增1；得到的值为43。第一个线程再从中断处继续执行；操作对象栈中的值还是42，因为那就是该线程执行GETFIELD时的lastId值。线程给lastId加1，得到43，存储这个结果。第一个线程也得到了值43。结果就是其中一个递增操作丢失了，因为第一个线程在被第二个线程打断后又踏入了第二个线程中。

将getNextId( )方法修改为同步方法就能修正这个问题。

A.2.3　小结

理解线程之间如何互相干涉，并不一定要精通字节码。如果你能看明白这个例子，它应该已经展示了多个线程之间互相干涉的可能性，这已经足够

了。

这个小例子说明，有必要尽量理解内存模型，明白什么是安全的，什么是不安全的。有一种普遍的误解，认为++（前递增或后递增）操作符是原子的，其实并非如此。你必须知道：

- 什么地方有共享对象/值；
- 哪些代码会导致并发读/写问题；
- 如何防止这种并发问题发生。

A.3　了解类库

A.3.1　Executor框架

如前文ExecutorClientScheduler.java所演示的那样，Java 5中引入的Executor框架支持利用线程池进行复杂的执行。那就是java.util.concurrent包中的一个类。

如果在创建线程时没有使用线程池或自行编写线程池，可以考虑使用Executor。它能让代码更整洁，易于理解，且更加短小。

Executor框架将把线程放到池中，自动调整其大小，并在必要时重建线程。它还支持future，一种通用的并发编程构造。Executor能与实现了Runnable的类协同工作，也能与实现了Callable接口的类协同工作。Callback看来就像是Runnable，但它能返回一个结果，那在多线程解决方案中是普遍的需求。

当代码需要执行多个相互独立的操作并等待这些操作结束时，future刚好就手：

```
public String processRequest(String message) throws Exception {
```

```
  Callable<String> makeExternalCall = new Callable<String>() {
    public String call() throws Exception {
      String result = "";
      // make external request
      return result;
    }
  };

  Future<String> result = executorService.submit(makeExternalCal
l);
  String partialResult = doSomeLocalProcessing();
  return result.get() + partialResult;
}
```

在本例中，方法开始执行makeExternalCall对象。然后该方法继续其他操作。最后一行代码调用result.get( )，在future代码执行完成前，这个操作是锁定的。

A.3.2　非锁定的解决方案

Java5虚拟机利用了现代处理器支持可靠、非锁定更新的设计优点。例如，考虑某个使用同步（从而也是锁定的）来提供线程安全地更新一个值的类：

```
public class ObjectWithValue {
  private int value;
  public void synchronized incrementValue() { ++value; }
  public int getValue() { return value; }
}
```

Java5有一系列用于此类情况的新类，例如AtomicBoolean、AtomicInteger和AtomicReference等；还有另外一些。我们可以重写上面的代码，使用非锁定的手段，如下所示：

```
public class ObjectWithValue {
  private AtomicInteger value = new AtomicInteger(0);

  public void incrementValue() {
    value.incrementAndGet();
  }

  public int getValue() {
    return value.get();
  }
}
```

即便使用了对象而非直接操作，使用了incrementAndGet( )这样的信息发送方式而非++操作，这个类的性能还是几乎总能胜过上一版本。在某些情况下只会快一点点，但较慢的情形却几乎不存在。

怎么会这样？现代处理器拥有一种通常称为比较交换（Compare and Swap，CAS）的操作。这种操作类似于数据库中的乐观锁定，而其同步版本则类似于保守锁定。

关键字synchronized总是要求上锁，即便第二

个线程并不更新同一值时也如此。尽管这种固有锁的性能一直在提升，但仍然代价昂贵。

非上锁的版本假定多个线程通常并不频繁修改同一个值，导致问题产生。它高效地侦测这种情形是否发生，并不断尝试，直至更新成功。这种侦测行为几乎总是比上锁来得划算，在争用激烈的情况下也是如此。

虚拟机如何实现这种机制？CAS的操作是原子的。逻辑上，CAS操作看起来像这样：

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
  int currentValue;
  do {
    currentValue = variableBeingSet
  } while(currentValue != compareAndSwap(currentValue, newValue)
);
}

int synchronized compareAndSwap(int currentValue, int newValue)
 {
  if(variableBeingSet == currentValue) {
    variableBeingSet = newValue;
    return currentValue;
  }
  return variableBeingSet;
}
```

当某个方法试图更新一个共享变量，CAS操作

就会验证要赋值的变量是否保有上一次的已知值。如果是，就修改变量值。如果不是，则不会碰变量，因为另一个线程正在试图更新变量值。要更新数据的方法（通过CAS操作）查看是否修改并持续尝试。

A.3.3　非线程安全类

有些类天生不是线程安全的。下面是几个例子：

- 数据库连接
- java.util中的容器
- Servlet

注意，有些群集类拥有一些线程安全的方法。不过，涉及调用多个方法的操作都不是线程安全的。例如，如果因为HashTable中已经有某物而不打算替换它，可能会写出以下代码：

```
if(!hashTable.containsKey(someKey)) {
        hashTable.put(someKey, new SomeValue());
}
```

单个方法是线程安全的。不过，另一个线程却可能在containsKey和put调用之间塞进一个值。有几

种修正这个问题的手段。

- 先锁定HashTable，确定其他使用者都做了基于客户端的锁定：

```
synchronized(map) {
if(!map.conainsKey(key))
   map.put(key,value);
}
```

- 用其对象包装HashTable，并使用不同的API——利用ADAPTER模式做基于服务端的锁定：

```
public class WrappedHashtable<K, V> {
  private Map<K, V> map = new Hashtable<K, V>();

  public synchronized void putIfAbsent(K key, V value) {
    if (map.containsKey(key))
        map.put(key, value);
  }
}
```

- 采用线程安全的群集：

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<
Integer, String>();
map.putIfAbsent(key, value);
```

在java.util.concurrent中的群集都有putIfAbsent(

)之类提供这种操作的方法。

A.4　方法之间的依赖可能破坏并发代码

以下是一个有关在方法间引入依赖的小例子：

```
public class IntegerIterator implements Iterator<Integer>
  private Integer nextValue = 0;

  public synchronized boolean hasNext() {
    return nextValue < 100000;
  }

  public synchronized Integer next() {
    if (nextValue == 100000)
      throw new IteratorPastEndException();
    return nextValue++;
  }

  public synchronized Integer getNextValue() {
    return nextValue;
  }
}
```

下面是使用IntegerIterator的代码：

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
  int nextValue = iterator.next();
  // do something with nextValue
}
```

如果只有一个线程执行这段代码，不会有什么问题。但如果有两个线程抱着每个线程都处理它获得的值、但列表中的每个元素都只被处理一次的意图，尝试共享IntegerIterator的单个实体，会发生什么事？多数时候什么也不会发生；线程开心地共享着列表，处理从迭代器获取的元素，在迭代器完成执行时停下。然而，在迭代的末尾，两个线程也有少量可能互相干涉，导致其中一个超出迭代器末尾，抛出异常。

问题在这里。线程1调用hasNext( )方法，该方法返回true。线程1占先，然后线程2也调用这个方法，同样返回true。线程2接着调用next( )，该方法如期返回一个值，但副作用是之后再调用hasNext( )就会返回false。线程1继续执行，以为hasNext( )还是true，然后调用next( )。即便单个方法是同步的，客户端还是使用了两个 方法。

这的确是个问题，也是并发代码中此类问题的典型例子。在这个特殊例子中，问题尤其隐蔽，因为只有在迭代器最后一次迭代时发生才会导致错误。如果线程刚好在那个点中断，其中一个线程就可能超出迭代器末尾。这类错误往往在系统部署之后很久才发生，而且很难追踪。

出现错误时，你有3种做法。

- 容忍错误；
- 修改客户代码解决问题：基于客户代码的锁定；
- 修改服务端代码解决问题，同时也修改了客户代码：基于服务端的锁定。

A.4.1　容忍错误

有时，可以通过一些设置让错误不会导致损害。例如，上述客户代码可以捕捉并清理异常。坦白地说，这有点草草从事，就像是半夜重启解决内存泄露问题一样。

A.4.2　基于客户代码的锁定

要让IntegerIterator在多线程情况下正确运行，对客户代码做如下修改：

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
  int nextValue;
  synchronized (iterator) {
    if (!iterator.hasNext())
      break;
    nextValue = iterator.next();
  }
  doSometingWith(nextValue);
}
```

每个客户端都通过synchronized关键字引入一个锁。这种重复违反了DRY原则，但如果代码使用非线程安全的第三方工具，可能必须这样做。

这种策略有风险，因为使用服务端的程序员都得记住在使用前上锁、用过后解锁。许多（许多！）年前，我遇到过一个在共享资源上应用基于客户代码锁定的系统。代码中有几百处用到这个资源的地方。有位可怜的程序员忘记在其中一处做资源锁定。

该系统是个多终端分时系统，为Local 705卡车司机联盟运行会计软件。计算机放在距Local 705总部50英里（约84.65km）以北的一间镶有高于地面的地板、环境可控的机房中。总部有几十位数据录入员，往终端输入记录。终端使用电话专线和600bit/s的半双工调制解调器连接到计算机。（这可是很久很久以前的事了。）

每天大概都会有一台终端毫无理由地“死锁”。死锁也不限定在某些终端或特定时间。就像是有人掷骰子选择死锁的时机和终端一般。有时，会有几台终端死锁。有时，好几天都不出现死锁情况。

刚开始，唯一的解决手段就是重启。但协同起来很不便。我们得打电话给总部，让大家都完成在

终端上的工作。然后我们才能关机、重启。如果有人在做要花上一两个小时才能做完的事，被锁定的终端就只能一直等着。

经过几个星期的调试，我们发现，原因在于一个指针不同步的环形缓冲区计数器。该缓冲区控制向终端的输出。指针值说明缓冲区是空的，但计数器却指出缓冲区是满的。因为缓冲区是空的，就没什么可显示；但因为缓冲区也是满的，也就无法向其中加入可在屏幕上显示的内容。

我们知道了终端为何会死锁，但却不知道为什么环形缓冲区会不同步。我们用了点手段发现问题所在。当时程序能够读取计算机的前面板开关状态（这可是很久很久以前的事了）。我们写了个陷阱程序，侦测这些开关何时被拨动，然后查找既空又满的环形缓冲区。如果找到，就重置该缓冲区为空。乌拉！锁定的终端又重新开始显示了。

这样，在终端锁定时就不必重启系统了。客户只需要打电话告诉我们出现死锁，我们就径直走到机房，拨动一下开关即可。

当然，有时他们会在周末加班，但是我们可不加班。所以我们又在计划列表中添加了一个函数，每分钟检查一次全部环形缓冲区，重置既空又满的

缓冲区。在客户打电话之前，显示就已经恢复正常了。

在发现问题原因之前，我们花了好几个星期查看一页又一页的单片机汇编语言代码。我们已经完成计算，算出死锁的频率是周期性的，而且其中有一处未受保护的环形缓冲区使用。所以，剩下的任务就是找出那个错误的用法。不幸这是多年以前的事，那时既没有搜索工具，也没有交叉引用或任何其他自动化帮助手段。我们只能细查代码清单。

在芝加哥1971年的寒冬，我学到了重要的一课。基于客户代码的锁定实在不可靠。

A.4.3　基于服务端的锁定

按照以下方式修改IntegerIterator也能消除重复：

```
public class IntegerIteratorServerLocked {
  private Integer nextValue = 0;
  public synchronized Integer getNextOrNull() {
    if (nextValue < 100000)
      return nextValue++;
    else
      return null;
  }
}
```

客户代码也要修改：

```
while (true) {
  Integer nextValue = iterator.getNextOrNull();
  if (next == null)
    break;
  // do something with nextValue
}
```

在这种情形下，我们实际上是修改了类的API，使其能适应多线程 [3] 。客户端需要做null检查，而不是检查hasNext( )。

通常你应该选用基于服务端的锁定，因为：

- 它减少了重复代码——采用基于客户代码的锁定，每个客户端都要正确锁定服务端。把锁定代码放到服务端，客户端就能自由使用对象，不必费心编写额外的锁定代码；
- 它提升了性能——在单线程部署中，可以用非多线程安全服务端代码替代线程安全客户端，从而省去花销；
- 它减少了出错的可能性——只会有一个程序员忘记上锁；
- 它执行了单一策略——该策略只在服务端这一处地方实施，而不是在许多地方（每个客户端）实施；

- 它缩减了共享变量的作用范围——客户端不必关心它们或它们是如何锁定的。一切都隐藏在服务端。如果出错，要侦查的范围就小多了。

如果你无法修改服务端代码又该如何？

- 使用ADAPTER模式修改API，添加锁定；

```
public class ThreadSafeIntegerIterator {
  private IntegerIterator iterator = new IntegerIterator();

  public synchronized Integer getNextOrNull() {
    if(iterator.hasNext())
      return iterator.next();
    return null;
  }
}
```

- 更好的方法是使用线程安全的群集和扩展接口。

A.5 提升吞吐量

假设我们打算连接上网，从一个URL列表中读取一组页面的内容。读到一个页面时，解析该页面并得到一些统计结果。读完所有页面后，打印出一份提要报表。

下面的类返回给定URL的页面内容：

```
public class PageReader {
  //...
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);

    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

下一个类是给出URL迭代器中每个页面的内容的迭代器：

```
public class PageIterator {
  private PageReader reader;
```

```
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
    if (urls.hasNext())
      getPageFor(urls.next());
    else
      return null;
  }

  public String getPageFor(String url) {
    return reader.getPageFor(url);
  }
}
```

PageIterator的一个实体可为多个不同线程共享，每个线程使用自己的PageReader实体读取并解析从迭代器中得到的页面。

注意，我们把synchronized代码块的数量限制在小范围之内。它只包括深处于PageIterator内部的临界区。最好是尽可能少地使用同步。

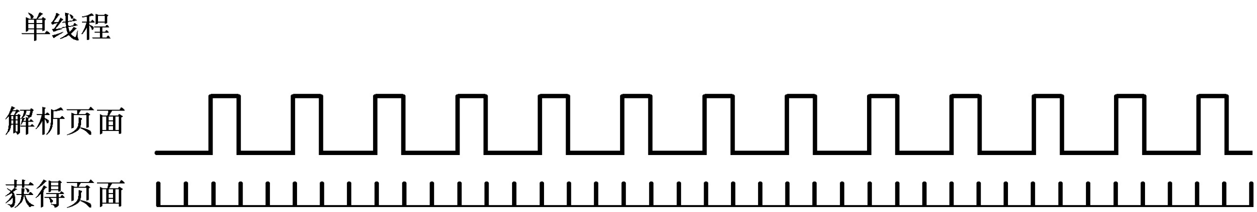
A.5.1 单线程条件下的吞吐量

来做个简单计算。鉴于讨论的目的，假定：

- 获取一个页面的I/O时间（平均）是1s；

- 解析一个页面的处理时间（平均）是0.5s；
- I/O操作不耗费处理器能力，而解析页面耗费100%处理器能力。

对于单个线程要处理的N个页面，总的执行时间为1.5s*N。图A-1显示了13个页面或大概19.5s的快照。



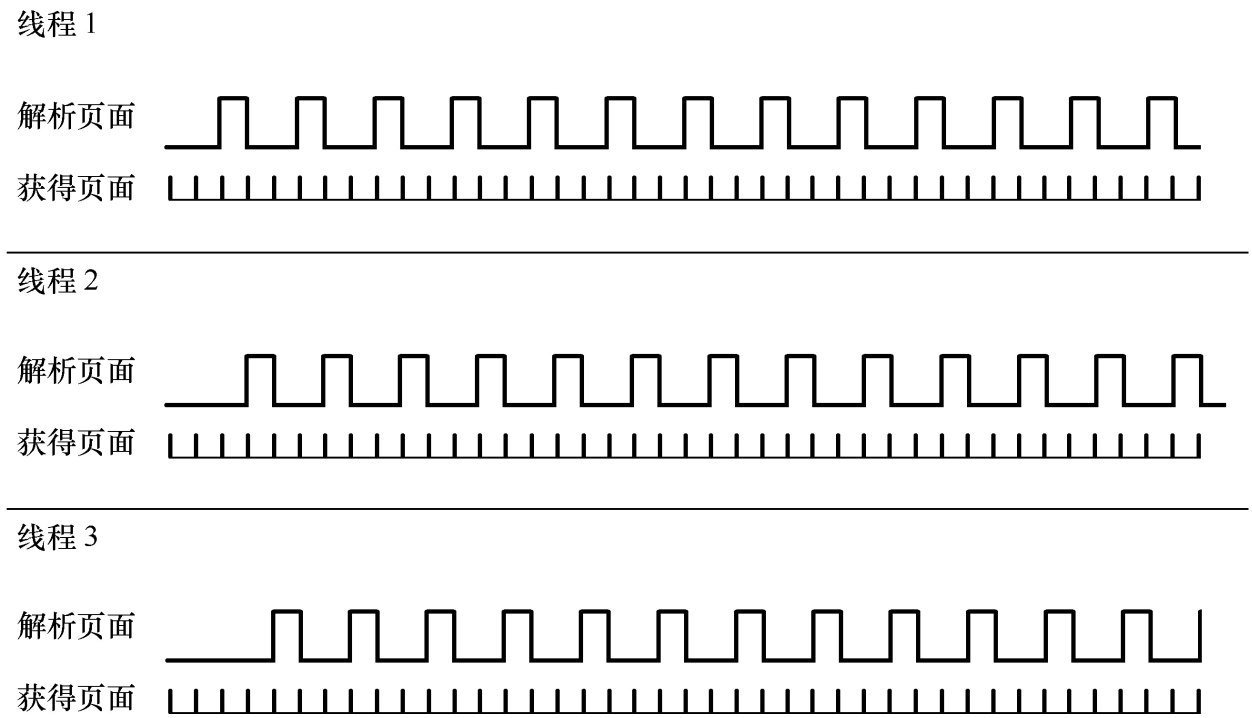
图A-1　单线程

A.5.2　多线程条件下的吞吐量

如果能够以任意次序获得页面并独立处理页面，就有可能利用多线程提升吞吐量。如果我们使用三个线程会如何？在同一时间内能获取多少个页面呢？

如你在图A-2中所见，多线程方案中与处理器能力有关的页面解析操作可以和与I/O有关的页面读取操作叠加进行。在理想状态下，这意味着处理器力尽其用。每个耗时一秒钟的页面读取操作都与两次解析操作叠加进行。这样，我们就能在每秒钟

内处理两个页面，即三倍于单线程方案的吞吐量。



图A-2　三个并发线程

A.6　死锁

想象一个拥有两个有限共享资源池的Web应用程序。

- 一个用于本地临时工作存储的数据库连接池；
- 一个用于连接到主存储库的MQ池。

假定该应用中有两个操作：创建和更新。

- 创建——获取到主存储库和数据库的连接。与主存储库协调，并把工作保存到本地临时工作数据库；
- 更新——先获取到数据库的连接，再获取到主存储库的连接。从临时工作数据库中读取数据，再发送给主存储库。

如果用户数量多于池的大小会怎样？假设每个池中能容纳10个资源。

- 有10个用户尝试创建，获取了10个数据库连接，每个线程在获取到数据库连接之后、获取到主存储库连接之前都被打断；
- 有10个用户尝试更新，获取了10个主存储库连接，每个线程在获取到主存储库连接之后、获

取到数据库连接之前都会被打断；

- 现在那10个“创建”线程必须等待获取主存储库连接，但那10个“更新”线程必须等待获取数据库连接；
- 死锁。系统永远无法恢复。

这听起来不太会出现，但谁会想要一个每隔一周就僵在那里不动的系统呢？谁想要调试出现了难以复现的症状的系统呢？这种问题突然发生，然后得花上好几个星期才能解决。

典型的“解决方案”是加入调试语句，发现问题。当然，调试语句对代码的修改足以令死锁在不同情况下发生，而且要几个月后才会再出现[4]。

要真正地解决死锁问题，我们需要理解死锁的原因。死锁的发生需要4个条件：

- 互斥；
- 上锁及等待；
- 无抢先机制；
- 循环等待。

A.6.1　互斥

当多个线程需要使用同一资源，且这些资源满

足下列条件时，互斥就会发生。

- 无法在同一时间为多个线程所用；
- 数量上有限制。

这种资源的常见例子是数据库连接、打开后用于写入的文件、记录锁或是信号量。

A.6.2　上锁及等待

当某个线程获取一个资源，在获取到其他全部所需资源并完成其工作之前，不会释放这个资源。

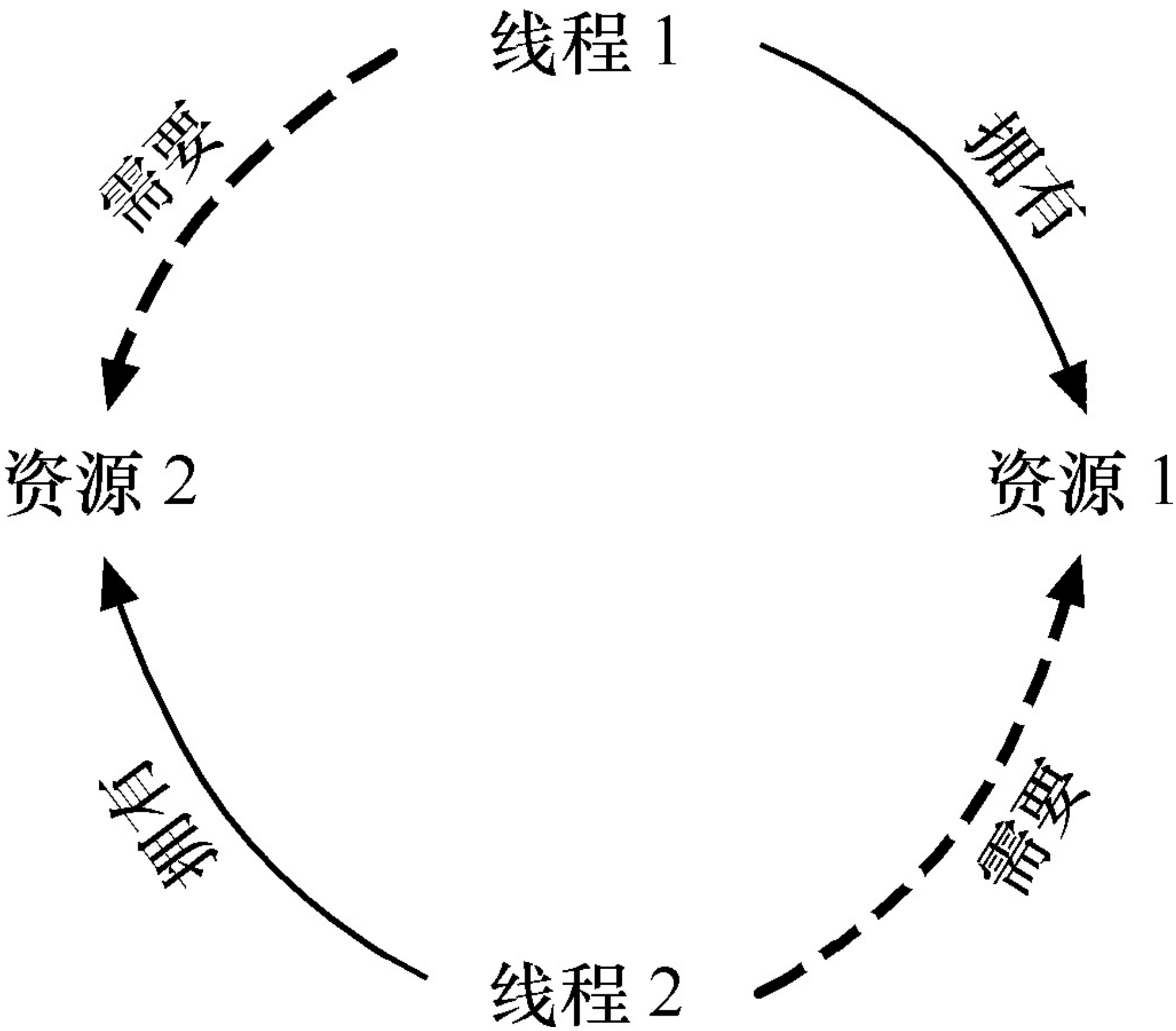
A.6.3　无抢先机制

线程无法从其他线程处夺取资源。一个线程持有资源时，其他线程获得这个资源的唯一手段就是等待该线程释放资源。

A.6.4　循环等待

这也被称为“死命拥抱”。想象两个线程，T1和T2，还有两个资源，R1和R2。T1拥有R1，T2拥有R2。T1需要R2，T2需要R1。如此就出现了如图A-3所示的情形。

这4种条件都是死锁所必需的。只要其中一个不满足，死锁就不会发生。



图A-3　循环等待

A.6.5　不互斥

避免死锁的一种策略是规避互斥条件。你可以：

- 使用允许同时使用的资源，如AtomicInteger；
- 增加资源数量，使其等于或大于竞争线程的数

量；

- 在获取资源之前，检查是否可用。

不幸的是，多数资源都有上限，且不能同时使用。而且第二个资源的标识也常常要依据对第一个资源的操作结果来判断。不过别丧气，还有3个其他条件呢。

A.6.6　不上锁及等待

如果拒绝等待，就能消除死锁。在获得资源之前检查资源，如果遇到某个繁忙资源，就释放所有资源，重新来过。

这种手段带来几个潜在问题：

- 线程饥饿——某个线程一直无法获得它所需的资源（它可能需要某种很少能同时获得的资源组合）；
- 活锁——几个线程可能会前后相连地要求获得某个资源，然后再释放一个资源，如此循环。这在单纯的CPU任务排列算法中尤其有可能出现（想想嵌入式设备或单纯的手写线程平衡算法）。

二者都能导致较差的吞吐量。第一个的结果是

CPU利用率低，第二个的结果是较高但无用的CPU利用率。

尽管这种策略听起来没效率，但也好过没有。至少，如果其他方案不奏效，这种手段几乎总可以用上。

A.6.7　满足抢先机制

避免死锁的另一策略是允许线程从其他线程上夺取资源。这通常利用一种简单的请求机制来实现。当线程发现资源繁忙，就要求其拥有者释放之。如果拥有者还在等待其他资源，就释放全部资源并重新来过。

这和上一种手段相似，但好处是允许线程等待资源。这减少了线程重新启动的次数。不过，管理所有请求可要花点心思。

A.6.8　不做循环等待

这是避免死锁的最常用手段。对于多数系统，它只要求一个为各方认同的约定。

在上面的例子中线程1同时需要资源1和资源2、线程2同时需要资源2和资源1，只要强制线程1

和线程2以同样次序分配资源，循环等待就不会发生。

更普遍地，如果所有线程都认同一种资源获取次序，并按照这种次序获取资源，死锁就不会发生。就像其他策略一样，这也会有问题：

- 获取资源的次序可能与使用资源的次序不匹配；一开始获取的资源可能在最后才会用到。这可能导致资源不必要地被长时间锁定；
- 有时无法强求资源获取顺序。如果第二个资源的ID来自对第一个资源操作的结果，获取次序也无从谈起。

有许多避免死锁的方法。有些会导致饥饿，另外一些会导致对CPU能力的大量耗费和降低响应率。TANSTAAFL [5]！

将解决方案中与线程相关的部分分隔出来，再加以调整和试验，是获得判断最佳策略所需的洞见的正道。

A.7　测试多线程代码

怎么才能编写显示以下代码有错的测试呢？

```
01:  public class ClassWithThreadingProblem {
02:    int nextId;
03:
04:    public int takeNextId() {
05:      return nextId++;
06:    }
07:  }
```

下面是对能证明上列代码有错的测试的描述：

- 记住nextId的当前值；
- 创建两个线程，每个都调用takeNextId( )一次；
- 验证nextId比开始时大2；
- 持续运行，直至发现nextId只比开始时大1为止。

代码清单A-2展示了这样一个测试：

代码清单A-2　ClassWithThreadingProblemTest.java

```
01:  package example;
02:
03:  import static org.junit.Assert.fail;
04:
05:  import  org.junit.Test;
```

```
06:
07:  public class ClassWithThreadingProblemTest {
08:    @Test
09:    public void twoThreadsShouldFailEventually() throws Excep
tion {
10:      final ClassWithThreadingProblem classWithThreadingProbl
em
                    =  new  ClassWithThreadingProblem();
11:
12:      Runnable runnable = new Runnable() {
13:        public void run() {
14:          classWithThreadingProblem.takeNextId();
15:        }
16:      };
17:
18:      for (int i = 0; i < 50000; ++i) {
19:        int startingId = classWithThreadingProblem.lastId;
20:        int expectedResult = 2 + startingId;
21:
22:        Thread t1 = new Thread(runnable);
23:        Thread t2 = new Thread(runnable);
24:        t1.start();
25:        t2.start();
26:        t1.join();
27:        t2.join();
28:
29:        int  endingId  =  classWithThreadingProblem.lastId;
30:
31:        if  (endingId  !=  expectedResult)
32:          return;
33:      }
34:
35:      fail("Should  have  exposed  a  threading  issue  but  it
  did  not.");
36:    }
37:  }
```

表**A-3**　代码清单**A-2**的注解

代码	描述

行	
10	创建ClassWithThreadingProblem的单个实体。注意，必须使用final关键字，因为要在一个匿名内部类中用到它
12～16	创建一个匿名内部类，该类用到ClassWithThreadingProblem的单个实体
18	运行这段代码“足够多”次以展示代码失败，但不要多到“花太长时间”。这是种平衡行为；我们不想等太久。选择这个数字有点难——尽管我们稍后会看到能够极大地降低这个数字
19	记住开始时的值。这个测试试图证明ClassWithThreadingProblem中的代码有错误。如果测试通过，它就证明了这一点。如果测试失败，它就没能证明代码出错
20	我们期望最终值比当前值大2
22～23	创建两个线程，都使用我们在第12~16行创建的对象。这样两个线程就有可能用到ClassWithThreadingProblem的单个实体，互相干涉
24～25	开始运行两个线程
26～27	在检查结果之前等待两个线程结束
29	记录真实的最终值
31～32	endingId是否与期待值不一样？如果是，测试结束——我们已经证明了代码有错误。如果不是，再试一次
35	到达这一步，测试无法证明产品代码在“合理范围”的时间内出错；测试失败了。要么是代码没错，要么是没有运行足够多次，错误条件还没满足

这个测试当然设置了满足并发更新问题发生的条件。不过，问题发生得如此频繁，测试也就极有可能侦测不到。

实际上，要真正侦测到问题，需要将循环数量设置到100万次以上。即便是这样，在10个100万次循环的执行中，错误也只发生了一次。这意味着我们可能要把循环次数设置为超过亿次才能获得可靠的失败证明。要等多久呢？

即便我们调优测试，在单台机器上得到可靠的失败证明，我们可能还需要用不同的值来重新设置测试，得到在其他机器、操作系统或不同版本的JVM上的失败证明。

而且这只是个简单 问题。如果连这个简单问题都无法轻易获得出错证明，我们怎么能真正侦测复杂问题呢？

我们能用什么手段来证明这个简单错误呢？而且，更重要的是，我们如何能写出证明更复杂代码中的错误的测试呢？我们怎样才能在不知道从何处着手时知道代码是否出错了呢？

下面是一些想法：

- 蒙特卡洛测试 。测试要灵活，便于调整。多次

运行测试——在一台测试服务器上——随机改变调整值。如果测试失败，代码就有错。确保及早编写这些测试，好让持续集成服务器尽快开始运行测试。另外，确认小心记录了在何种条件下测试失败。

- 在每种目标部署平台上运行测试。重复运行。持续运行。测试在不失败的前提下运行得越久，就越能说明：
  - 生产代码正确；

    或；
  - 测试不足以暴露问题。
- 在另一台有不同负载的机器上运行测试。能模拟生产环境的负载，就模拟之。
- 即便你做了所有这些，还是不见得有很好的机会发现代码中的线程问题。最阴险的问题拥有很小的截面，在十亿次执行中只会发生一次。这类错误是复杂系统的噩梦。

A.8　测试线程代码的工具支持

IBM提供了一个名为ConTest的工具[6]。它能对类进行装置，令非线程安全代码更有可能失败。

我们与IBM或开发ConTest的团队没有直接关系。有位同事发现了这个工具。在用了几分钟后，我们发现自己发现线程问题的能力得到了很大提升。

下面是使用ConTest的简要步骤：

- 编写测试和生产代码，确保有专门模拟多用户在多种负载情况下操作的测试，如上文所述；
- 用ConTest装置测试和生产代码；
- 运行测试。

用ConTest装置代码后，原本千万次循环才能暴露一个错误的比率提升到30次循环就能找到错误。以下是装置代码后的几次测试运行结果值：13、23、0、54、16、14、6、69、107、49和2。显然装置后的类更加容易和可靠地被证明失败。

A.9　小结

本章只是在并发编程广阔而可怕的领地上的短暂逗留罢了。我们只触及了地表。我们在这里强调的，只是保持并发代码整洁的一些规程，如果要编写并发系统，还有许多东西要学。建议从Doug Lea的大作*Concurrent Programming in Java: Design Principles and Patterns* 开始[7]。

在本章中，我们谈到并发更新，还有清理及避免同步的规程。我们谈到线程如何提升与I/O有关的系统的吞吐量，展示了获得这种提升的整洁技术。我们谈到死锁及干净地避免死锁的规程。最后，我们谈到通过装置代码暴露并发问题的策略。

A.10 教程：完整代码范例

A.10.1 客户端/服务器非线程代码

代码清单A-3 Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
  ServerSocket serverSocket;
  volatile boolean keepProcessing = true;

  public Server(int port, int millisecondsTimeout) throws IOExce
ption {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(millisecondsTimeout);
  }

  public void run() {
    System.out.printf("Server Starting\n");

    while (keepProcessing) {
      try {
        System.out.printf("accepting client\n");
        Socket socket = serverSocket.accept();
        System.out.printf("got client\n");
        process(socket);
      } catch (Exception e) {
        handle(e);
      }
    }
```

```
  }

  private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
      e.printStackTrace();
    }
  }

  public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
  }

  void process(Socket socket) {
    if (socket == null)
      return;

    try {
      System.out.printf("Server: getting message\n");
      String message = MessageUtils.getMessage(socket);
      System.out.printf("Server: got message: %s\n", message);
      Thread.sleep(1000);
      System.out.printf("Server: sending reply: %s\n", message);
      MessageUtils.sendMessage(socket, "Processed: " + message);
      System.out.printf("Server: sent\n");
      closeIgnoringException(socket);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void closeIgnoringException(Socket socket) {
    if (socket != null)
      try {
          socket.close();
      } catch (IOException ignore) {
      }
  }

  private void closeIgnoringException(ServerSocket serverSocket)
 {
    if (serverSocket != null)
      try {
          serverSocket.close();
      } catch (IOException ignore) {
```

```
    }
  }
}
```

代码清单A-4　ClientTest.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
  ServerSocket serverSocket;
  volatile boolean keepProcessing = true;
  public Server(int port, int millisecondsTimeout) throws IOExce
ption {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(millisecondsTimeout);
  }

  public void run() {
    System.out.printf("Server Starting\n");

    while (keepProcessing) {
      try {
        System.out.printf("accepting client\n");
        Socket socket = serverSocket.accept();
        System.out.printf("got client\n");
        process(socket);
      } catch (Exception e) {
        handle(e);
      }
    }
  }

  private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
      e.printStackTrace();
```

```
    }
  }

  public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
  }

  void process(Socket socket) {
    if (socket == null)
      return;

    try {
      System.out.printf("Server: getting message\n");
      String message = MessageUtils.getMessage(socket);
      System.out.printf("Server: got message: %s\n", message);
      Thread.sleep(1000);
      System.out.printf("Server: sending reply: %s\n", message);
      MessageUtils.sendMessage(socket, "Processed: " + message);
      System.out.printf("Server: sent\n");
      closeIgnoringException(socket);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void closeIgnoringException(Socket socket) {
    if (socket != null)
      try {
        socket.close();
      } catch (IOException ignore) {
      }
  }

  private void closeIgnoringException(ServerSocket serverSocket)
 {
    if (serverSocket != null)
      try {
        serverSocket.close();
      } catch (IOException ignore) {
      }
  }
}
```

代码清单A-5　MessageUtils.java

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
  public static void sendMessage(Socket socket, String message)
            throws IOException {
    OutputStream stream = socket.getOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(stream);
    oos.writeUTF(message);
    oos.flush();
  }

  public static String getMessage(Socket socket) throws IOExcept
ion {
    InputStream stream = socket.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(stream);
    return ois.readUTF();
  }
}
```

A.10.2　使用线程的客户端/服务器代码

把服务器修改为使用多线程，只需要对处理消息进行修改即可（新的代码行用粗体标出）：

```
void process(final Socket socket) {
  if (socket == null)
    return;
```

```
  Runnable clientHandler = new Runnable() {



    public void run() {
      try {
        System.out.printf("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message)
;
        MessageUtils.sendMessage(socket, "Processed: " + message)
;
        System.out.printf("Server: sent\n");
        closeIgnoringException(socket);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  };

 Thread clientConnection = new Thread(clientHandler);



  clientConnection.start();



}
```

---

[1] 原注：你可以自行验证修改之前和之后的代码。复查前文的非多线程代码。复查之后的多线程代码。

[2]　原注：这说得有点简单了。鉴于讨论的目的，我们就用这个简化模型好了。

[3]　原注：实际上，Iterator接口天生不是线程安全的。它并不为多线程而设计，所以出现这种情况也不奇怪。

[4]　原注：例如，有人添加了一些调试输出，问题“不见了”。调试代码“修正”了问题，其实问题还在系统中存在。

[5]　原注：世上没有免费的午餐（There ain’t no such thing as a free lunch）。

[6]　原注：http://www.haifa.ibm.com/projects/verification/co
。

[7]　原注：见[Lea99]p.191。

附录B　org.jfree.date.SerialDate

代码清单B-1　SerialDate.Java

```
   1 /* =============================================================
==================
   2  * JCommon : a free general purpose class library for the
Java(tm) platform
   3  * =============================================================
==================
   4  *
   5  * (C) Copyright 2000-2005, by Object Refinery Limited and
 Contributors.
   6  *
   7  * Project Info:  http://www.jfree.org/jcommon/index.html



   8  *
   9  * This library is free software; you can redistribute it
and/or modify it
  10  * under the terms of the GNU Lesser General Public Licens
e as published by
  11  * the Free Software Foundation; either version 2.1 of the
 License, or
  12  * (at your option) any later version.
  13  *
  14  * This library is distributed in the hope that it will be
 useful, but
  15  * WITHOUT ANY WARRANTY; without even the implied warranty
 of MERCHANTABILITY
  16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
 General Public
  17  * License for more details.
  18  *
  19  * You should have received a copy of the GNU Lesser Gener
al Public
  20  * License along with this library; if not, write to the F
ree Software
  21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Bost
on, MA  02110-1301,
```

```
  22  * USA.
  23  *
  24  * [Java is a trademark or registered trademark of Sun Mic
rosystems, Inc.
  25  * in the United States and other countries.]
  26  *
  27  * ---------------
  28  * SerialDate.java
  29  * ---------------
  30  * (C) Copyright 2001-2005, by Object Refinery Limited.
  31  *
  32  * Original Author:  David Gilbert (for Object Refinery Li
mited);
  33  * Contributor(s):   -;
  34  *
  35  * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady
Exp $
  36  *
  37  * Changes (from 11-Oct-2001)
  38  * --------------------------
  39  * 11-Oct-2001 : Re-organised the class and moved it to ne
w package
  40  *               com.jrefinery.date (DG);
  41  * 05-Nov-2001 : Added a getDescription() method, and elim
inated NotableDate
  42  *               class (DG);
  43  * 12-Nov-2001 : IBD requires setDescription() method, now
 that NotableDate
  44  *               class is gone (DG);  Changed getPreviousD
ayOfWeek(),
  45  *               getFollowingDayOfWeek() and getNearestDay
OfWeek() to correct
  46  *               bugs (DG);
  47  * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
  48  * 29-May-2002 : Moved the month constants into a separate
 interface
  49  *               (MonthConstants) (DG);
  50  * 27-Aug-2002 : Fixed bug in addMonths() method, thanks t
o N???levka Petr (DG);
  51  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
  52  * 13-Mar-2003 : Implemented Serializable (DG);
  53  * 29-May-2003 : Fixed bug in addMonths method (DG);
  54  * 04-Sep-2003 : Implemented Comparable.  Updated the isIn
Range javadocs (DG);
  55  * 05-Jan-2005 : Fixed bug in addYears() method (1096282)
```

```
(DG);
  56  *
  57  */
  58
  59 package org.jfree.date;
  60
  61 import java.io.Serializable;
  62 import java.text.DateFormatSymbols;
  63 import java.text.SimpleDateFormat;
  64 import java.util.Calendar;
  65 import java.util.GregorianCalendar;
  66
  67 /**
  68  *  An abstract class that defines our requirements for ma
nipulating dates,
  69  *  without tying down a particular implementation.
  70  *  <P>
  71  *  Requirement 1 : match at least what Excel does for dat
es;
  72  *  Requirement 2 : class is immutable;
  73  *  <P>
  74  *  Why not just use java.util.Date?  We will, when it mak
es sense.  At times,
  75  *  java.util.Date can be *too* precise - it represents an
 instant in time,
  76  *  accurate to 1/1000th of a second (with the date itself
 depending on the
  77  *  time-zone).  Sometimes we just want to represent a par
ticular day (e.g. 21
  78  *  January 2015) without concerning ourselves about the t
ime of day, or the
  79  *  time-zone, or anything else.  That's what we've define
d SerialDate for.
  80  *  <P>
  81  *  You can call getInstance() to get a concrete subclass
of SerialDate,
  82  *  without worrying about the exact implementation.
  83  *
  84  * @author David Gilbert
  85  */
  86 public abstract class SerialDate implements Comparable,
  87                                             Serializable,
  88                                             MonthConstants
 {
  89
```

```
  90     /** For serialization. */
  91     private static final long serialVersionUID = -29371604
0467423637L;
  92
  93     /** Date format symbols. */
  94     public static final DateFormatSymbols
  95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDa
teFormatSymbols();
  96
  97     /** The serial number for 1 January 1900. */
  98     public static final int SERIAL_LOWER_BOUND = 2;
  99
 100     /** The serial number for 31 December 9999. */
 101     public static final int SERIAL_UPPER_BOUND = 2958465;
 102
 103     /** The lowest year value supported by this date forma
t. */
 104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
 105
 106     /** The highest year value supported by this date form
at. */
 107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
 108
 109     /** Useful constant for Monday. Equivalent to java.uti
l.Calendar.MONDAY. */
 110     public static final int MONDAY = Calendar.MONDAY;
 111
 112     /**
 113      * Useful constant for Tuesday. Equivalent to java.uti
l.Calendar.TUESDAY.
 114      */
 115     public static final int TUESDAY = Calendar.TUESDAY;
 116
 117     /**
 118      * Useful constant for Wednesday. Equivalent to
 119      * java.util.Calendar.WEDNESDAY.
 120      */
 121     public static final int WEDNESDAY = Calendar.WEDNESDAY
;
 122
 123     /**
 124      * Useful constant for Thrusday. Equivalent to java.ut
il.Calendar.THURSDAY.
 125      */
 126     public static final int THURSDAY = Calendar.THURSDAY;
```

```
 127
 128     /** Useful constant for Friday. Equivalent to java.uti
l.Calendar.FRIDAY. */
 129     public static final int FRIDAY = Calendar.FRIDAY;
 130
 131     /**
 132      * Useful constant for Saturday. Equivalent to java.ut
il.Calendar.SATURDAY.
 133      */
 134     public static final int SATURDAY = Calendar.SATURDAY;
 135
 136     /** Useful constant for Sunday. Equivalent to java.uti
l.Calendar.SUNDAY. */
 137     public static final int SUNDAY = Calendar.SUNDAY;
 138
 139     /** The number of days in each month in non leap years
. */
 140     static final int[] LAST_DAY_OF_MONTH =
 141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
 142
 143     /** The number of days in a (non-leap) year up to the
end of each month. */
 144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
 145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304,
 334, 365};
 146
 147     /** The number of days in a year up to the end of the
preceding month. */
 148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_
MONTH =
 149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 3
04, 334, 365};
 150
 151     /** The number of days in a leap year up to the end of
 each month. */
 152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_
MONTH =
 153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305,
 335, 366};
 154
 155     /**
 156      * The number of days in a leap year up to the end of
the preceding month.
 157      */
```

```
 158     static final int[]
 159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH
 =
 160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 27
4, 305, 335, 366};
 161
 162     /** A useful constant for referring to the first week
in a month. */
 163     public static final int FIRST_WEEK_IN_MONTH = 1;
 164
 165     /** A useful constant for referring to the second week
 in a month. */
 166     public static final int SECOND_WEEK_IN_MONTH = 2;
 167
 168     /** A useful constant for referring to the third week
in a month. */
 169     public static final int THIRD_WEEK_IN_MONTH = 3;
 170
 171     /** A useful constant for referring to the fourth week
 in a month. */
 172     public static final int FOURTH_WEEK_IN_MONTH = 4;
 173
 174     /** A useful constant for referring to the last week i
n a month. */
 175     public static final int LAST_WEEK_IN_MONTH = 0;
 176
 177     /** Useful range constant. */
 178     public static final int INCLUDE_NONE = 0;
 179
 180     /** Useful range constant. */
 181     public static final int INCLUDE_FIRST = 1;
 182
 183     /** Useful range constant. */
 184     public static final int INCLUDE_SECOND = 2;
 185
 186     /** Useful range constant. */
 187     public static final int INCLUDE_BOTH = 3;
 188
 189     /**
 190      * Useful constant for specifying a day of the week re
lative to a fixed
 191      * date.
 192      */
 193     public static final int PRECEDING = -1;
 194
```

```
 195     /**
 196      * Useful constant for specifying a day of the week re
lative to a fixed
 197      * date.
 198      */
 199     public static final int NEAREST = 0;
 200
 201     /**
 202      * Useful constant for specifying a day of the week re
lative to a fixed
 203      * date.
 204      */
 205     public static final int FOLLOWING = 1;
 206
 207     /** A description for the date. */
 208     private String description;
 209
 210     /**
 211      * Default constructor.
 212      */
 213     protected SerialDate() {
 214     }
 215
 216     /**
 217      * Returns <code>true</code> if the supplied integer c
ode represents a
 218      * valid day-of-the-week, and <code>false</code> other
wise.
 219      *
 220      * @param code  the code being checked for validity.
 221      *
 222      * @return <code>true</code> if the supplied integer c
ode represents a
 223      *         valid day-of-the-week, and <code>false</cod
e> otherwise.
 224      */
 225     public static boolean isValidWeekdayCode(final int cod
e) {
 226
 227         switch(code) {
 228             case SUNDAY:
 229             case MONDAY:
 230             case TUESDAY:
 231             case WEDNESDAY:
 232             case THURSDAY:
```

```
 233               case FRIDAY:
 234               case SATURDAY:
 235                   return true;
 236               default:
 237                   return false;
 238           }
 239 
 240       }
 241 
 242       /**
 243        * Converts the supplied string to a day of the week.
 244        *
 245        * @param s  a string representing the day of the week
.
 246        *
 247        * @return <code>-1</code> if the string is not conver
table, the day of
 248        *         the week otherwise.
 249        */
 250       public static int stringToWeekdayCode(String s) {
 251 
 252           final String[] shortWeekdayNames
 253               = DATE_FORMAT_SYMBOLS.getShortWeekdays();
 254           final String[] weekDayNames = DATE_FORMAT_SYMBOLS.
getWeekdays();
 255 
 256           int result = -1;
 257           s = s.trim();
 258           for (int i = 0; i < weekDayNames.length; i++) {
 259               if (s.equals(shortWeekdayNames[i])) {
 260                   result = i;
 261                   break;
 262               }
 263               if (s.equals(weekDayNames[i])) {
 264                   result = i;
 265                   break;
 266               }
 267           }
 268           return result;
 269 
 270       }
 271 
 272       /**
 273        * Returns a string representing the supplied day-of-t
he-week.
```

```
 274      * <P>
 275      * Need to find a better approach.
 276      *
 277      * @param weekday  the day of the week.
 278      *
 279      * @return a string representing the supplied day-of-t
he-week.
 280      */
 281     public static String weekdayCodeToString(final int wee
kday) {
 282
 283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getW
eekdays();
 284         return weekdays[weekday];
 285
 286     }
 287
 288     /**
 289      * Returns an array of month names.
 290      *
 291      * @return an array of month names.
 292      */
 293     public static String[] getMonths() {
 294
 295         return getMonths(false);
 296
 297     }
 298
 299     /**
 300      * Returns an array of month names.
 301      *
 302      * @param shortened  a flag indicating that shortened
month names should
 303      *                   be returned.
 304      *
 305      * @return an array of month names.
 306      */
 307     public static String[] getMonths(final boolean shorten
ed) {
 308
 309         if (shortened) {
 310             return DATE_FORMAT_SYMBOLS.getShortMonths();
 311         }
 312         else {
 313             return DATE_FORMAT_SYMBOLS.getMonths();
```

```
 314         }
 315
 316     }
 317
 318     /**
 319      * Returns true if the supplied integer code represent
s a valid month.
 320      *
 321      * @param code  the code being checked for validity.
 322      *
 323      * @return <code>true</code> if the supplied integer c
ode represents a
 324      *         valid month.
 325      */
 326     public static boolean isValidMonthCode(final int code)
 {
 327
 328         switch(code) {
 329             case JANUARY:
 330             case FEBRUARY:
 331             case MARCH:
 332             case APRIL:
 333             case MAY:
 334             case JUNE:
 335             case JULY:
 336             case AUGUST:
 337             case SEPTEMBER:
 338             case OCTOBER:
 339             case NOVEMBER:
 340             case DECEMBER:
 341                 return true;
 342             default:
 343                 return false;
 344         }
 345
 346     }
 347
 348     /**
 349      * Returns the quarter for the specified month.
 350      *
 351      * @param code  the month code (1-12).
 352      *
 353      * @return the quarter that the month belongs to.
 354      * @throws java.lang.IllegalArgumentException
 355      */
```

```
 356     public static int monthCodeToQuarter(final int code) {
 357
 358         switch(code) {
 359             case JANUARY:
 360             case FEBRUARY:
 361             case MARCH: return 1;
 362             case APRIL:
 363             case MAY:
 364             case JUNE: return 2;
 365             case JULY:
 366             case AUGUST:
 367             case SEPTEMBER: return 3;
 368             case OCTOBER:
 369             case NOVEMBER:
 370             case DECEMBER: return 4;
 371             default: throw new IllegalArgumentException(
 372                 "SerialDate.monthCodeToQuarter: invalid mo
nth code.");
 373         }
 374
 375     }
 376
 377     /**
 378      * Returns a string representing the supplied month.
 379      * <P>
 380      * The string returned is the long form of the month n
ame taken from the
 381      * default locale.
 382      *
 383      * @param month  the month.
 384      *
 385      * @return a string representing the supplied month.
 386      */
 387     public static String monthCodeToString(final int month
) {
 388
 389         return monthCodeToString(month, false);
 390
 391     }
 392
 393     /**
 394      * Returns a string representing the supplied month.
 395      * <P>
 396      * The string returned is the long or short form of th
e month name taken
```

```
 397      * from the default locale.
 398      *
 399      * @param month  the month.
 400      * @param shortened  if <code>true</code> return the a
bbreviation of the
 401      *                   month.
 402      *
 403      * @return a string representing the supplied month.
 404      * @throws java.lang.IllegalArgumentException
 405      */
 406     public static String monthCodeToString(final int month
,
 407                                            final boolean s
hortened) {
 408
 409         // check arguments...
 410         if (!isValidMonthCode(month)) {
 411             throw new IllegalArgumentException(
 412                 "SerialDate.monthCodeToString: month outsi
de valid range.");
 413         }
 414
 415         final String[] months;
 416
 417         if (shortened) {
 418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
 419         }
 420         else {
 421             months = DATE_FORMAT_SYMBOLS.getMonths();
 422         }
 423
 424         return months[month - 1];
 425
 426     }
 427
 428     /**
 429      * Converts a string to a month code.
 430      *


<P>
 431      * This method will return one of the constants JANUAR
Y, FEBRUARY, ...,
 432      * DECEMBER that corresponds to the string.  If the st
ring is not
```

```
 433      * recognised, this method returns -1.
 434      *
 435      * @param s  the string to parse.
 436      *
 437      * @return <code>-1</code> if the string is not parsea
ble, the month of the
 438      *         year otherwise.
 439      */
 440     public static int stringToMonthCode(String s) {
 441
 442         final String[] shortMonthNames = DATE_FORMAT_SYMBO
LS.getShortMonths();
 443         final String[] monthNames = DATE_FORMAT_SYMBOLS.ge
tMonths();
 444
 445         int result = -1;
 446         s = s.trim();
 447
 448         // first try parsing the string as an integer (1-1
2)...
 449         try {
 450             result = Integer.parseInt(s);
 451         }
 452         catch (NumberFormatException e) {
 453             // suppress
 454         }
 455
 456         // now search through the month names...
 457         if ((result < 1) || (result > 12)) {
 458             for (int i = 0; i < monthNames.length; i++) {
 459                 if (s.equals(shortMonthNames[i])) {
 460                     result = i + 1;
 461                     break;
 462                 }
 463                 if (s.equals(monthNames[i])) {
 464                     result = i + 1;
 465                     break;
 466                 }
 467             }
 468         }
 469
 470         return result;
 471
 472     }
 473
```

```
 474     /**
 475      * Returns true if the supplied integer code represent
s a valid
 476      * week-in-the-month, and false otherwise.
 477      *
 478      * @param code  the code being checked for validity.
 479      * @return <code>true</code> if the supplied integer c
ode represents a
 480      *         valid week-in-the-month.
 481      */
 482     public static boolean isValidWeekInMonthCode(final int
 code) {
 483
 484         switch(code) {
 485             case FIRST_WEEK_IN_MONTH:
 486             case SECOND_WEEK_IN_MONTH:
 487             case THIRD_WEEK_IN_MONTH:
 488             case FOURTH_WEEK_IN_MONTH:
 489             case LAST_WEEK_IN_MONTH: return true;
 490             default: return false;
 491         }
 492
 493     }
 494
 495     /**
 496      * Determines whether or not the specified year is a l
eap year.
 497      *
 498      * @param yyyy  the year (in the range 1900 to 9999).
 499      *
 500      * @return <code>true</code> if the specified year is
a leap year.
 501      */
 502     public static boolean isLeapYear(final int yyyy) {
 503
 504         if ((yyyy % 4) != 0) {
 505             return false;
 506         }
 507         else if ((yyyy % 400) == 0) {
 508             return true;
 509         }
 510         else if ((yyyy % 100) == 0) {
 511             return false;
 512         }
 513         else {
```

```
 514                return true;
 515            }
 516
 517        }
 518
 519        /**
 520         * Returns the number of leap years from 1900 to the s
pecified year
 521         * INCLUSIVE.
 522         * <P>
 523         * Note that 1900 is not a leap year.
 524         *
 525         * @param yyyy  the year (in the range 1900 to 9999).
 526         *
 527         * @return the number of leap years from 1900 to the s
pecified year.
 528         */
 529        public static int leapYearCount(final int yyyy) {
 530
 531            final int leap4 = (yyyy - 1896) / 4;
 532            final int leap100 = (yyyy - 1800) / 100;
 533            final int leap400 = (yyyy - 1600) / 400;
 534            return leap4 - leap100 + leap400;
 535
 536        }
 537
 538        /**
 539         * Returns the number of the last day of the month, ta
king into account
 540         * leap years.
 541         *
 542         * @param month  the month.
 543         * @param yyyy  the year (in the range 1900 to 9999).
 544         *
 545         * @return the number of the last day of the month.
 546         */
 547        public static int lastDayOfMonth(final int month, fina
l int yyyy) {
 548
 549            final int result = LAST_DAY_OF_MONTH[month];
 550            if (month != FEBRUARY) {
 551                return result;
 552            }
 553            else if (isLeapYear(yyyy)) {
 554                return result + 1;
```

```
 555         }
 556         else {
 557             return result;
 558         }
 559
 560     }
 561
 562     /**
 563      * Creates a new date by adding the specified number o
f days to the base
 564      * date.
 565      *
 566      * @param days  the number of days to add (can be nega
tive).
 567      * @param base  the base date.
 568      *
 569      * @return a new date.
 570      */
 571     public static SerialDate addDays(final int days, final
 SerialDate base) {
 572
 573         final int serialDayNumber = base.toSerial() + days
;
 574         return SerialDate.createInstance(serialDayNumber);
 575
 576     }
 577
 578     /**
 579      * Creates a new date by adding the specified number o
f months to the base
 580      * date.
 581      * <P>
 582      * If the base date is close to the end of the month,
the day on the result
 583      * may be adjusted slightly:  31 May + 1 month = 30 Ju
ne.
 584      *
 585      * @param months  the number of months to add (can be
negative).
 586      * @param base  the base date.
 587      *
 588      * @return a new date.
 589      */
 590     public static SerialDate addMonths(final int months,
 591                                        final SerialDate ba
```

```
se) {
 592
 593         final int yy = (12 * base.getYYYY() + base.getMont
h() + months - 1)
 594                        / 12;
 595         final int mm = (12 * base.getYYYY() + base.getMont
h() + months - 1)
 596                        % 12 + 1;
 597         final int dd = Math.min(
 598             base.getDayOfMonth(), SerialDate.lastDayOfMont
h(mm, yy)
 599         );
 600         return SerialDate.createInstance(dd, mm, yy);
 601
 602     }
 603
 604     /**
 605      * Creates a new date by adding the specified number o
f years to the base
 606      * date.
 607      *
 608      * @param years  the number of years to add (can be ne
gative).
 609      * @param base  the base date.
 610      *
 611      * @return A new date.
 612      */
 613     public static SerialDate addYears(final int years, fin
al SerialDate base) {
 614
 615         final int baseY = base.getYYYY();
 616         final int baseM = base.getMonth();
 617         final int baseD = base.getDayOfMonth();
 618
 619         final int targetY = baseY + years;
 620         final int targetD = Math.min(
 621             baseD, SerialDate.lastDayOfMonth(baseM, target
Y)
 622         );
 623
 624         return SerialDate.createInstance(targetD, baseM, t
argetY);
 625
 626     }
 627
```

```
 628     /**
 629      * Returns the latest date that falls on the specified
 day-of-the-week and
 630      * is BEFORE the base date.
 631      *
 632      * @param targetWeekday  a code for the target day-of-
the-week.
 633      * @param base  the base date.
 634      *
 635      * @return the latest date that falls on the specified
 day-of-the-week and
 636      *         is BEFORE the base date.
 637      */
 638     public static SerialDate getPreviousDayOfWeek(final in
t targetWeekday,
 639                                                    final Se
rialDate base) {
 640
 641         // check arguments...
 642         if (!SerialDate.isValidWeekdayCode(targetWeekday))
 {
 643             throw new IllegalArgumentException(
 644                 "Invalid day-of-the-week code."
 645             );
 646         }
 647
 648         // find the date...
 649         final int adjust;
 650         final int baseDOW = base.getDayOfWeek();
 651         if (baseDOW > targetWeekday) {
 652             adjust = Math.min(0, targetWeekday - baseDOW);
 653         }
 654         else {
 655             adjust = -7 + Math.max(0, targetWeekday - base
DOW);



 656         }
 657
 658         return SerialDate.addDays(adjust, base);
 659
 660     }
 661
 662     /**
```

```
 663      * Returns the earliest date that falls on the specifi
ed day-of-the-week
 664      * and is AFTER the base date.
 665      *
 666      * @param targetWeekday  a code for the target day-of-
the-week.
 667      * @param base  the base date.
 668      *
 669      * @return the earliest date that falls on the specifi
ed day-of-the-week
 670      *         and is AFTER the base date.
 671      */
 672     public static SerialDate getFollowingDayOfWeek(final i
nt targetWeekday,
 673                                                     final S
erialDate base) {
 674
 675         // check arguments...
 676         if (!SerialDate.isValidWeekdayCode(targetWeekday))
 {
 677             throw new IllegalArgumentException(
 678                 "Invalid day-of-the-week code."
 679             );
 680         }
 681
 682         // find the date...
 683         final int adjust;
 684         final int baseDOW = base.getDayOfWeek();
 685         if (baseDOW > targetWeekday) {
 686             adjust = 7 + Math.min(0, targetWeekday - baseD
OW);
 687         }
 688         else {
 689             adjust = Math.max(0, targetWeekday - baseDOW);
 690         }
 691
 692         return SerialDate.addDays(adjust, base);
 693     }
 694
 695     /**
 696      * Returns the date that falls on the specified day-of
-the-week and is
 697      * CLOSEST to the base date.
 698      *
 699      * @param targetDOW  a code for the target day-of-the-
```

```
week.
 700      * @param base  the base date.
 701      *
 702      * @return the date that falls on the specified day-of
-the-week and is
 703      *         CLOSEST to the base date.
 704      */
 705     public static SerialDate getNearestDayOfWeek(final int
 targetDOW,
 706                                                   final Ser
ialDate base) {
 707
 708         // check arguments...
 709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
 710             throw new IllegalArgumentException(
 711                 "Invalid day-of-the-week code."
 712             );
 713         }
 714
 715         // find the date...
 716         final int baseDOW = base.getDayOfWeek();
 717         int adjust = -Math.abs(targetDOW - baseDOW);
 718         if (adjust >= 4) {
 719             adjust = 7 - adjust;
 720         }
 721         if (adjust <= -4) {
 722             adjust = 7 + adjust;
 723         }
 724         return SerialDate.addDays(adjust, base);
 725
 726     }
 727
 728     /**
 729      * Rolls the date forward to the last day of the month
.
 730      *
 731      * @param base  the base date.
 732      *
 733      * @return a new serial date.
 734      */
 735     public SerialDate getEndOfCurrentMonth(final SerialDat
e base) {
 736         final int last = SerialDate.lastDayOfMonth(
 737             base.getMonth(), base.getYYYY()
 738         );
```

```
 739         return SerialDate.createInstance(last, base.getMon
th(), base.getYYYY());
 740     }
 741
 742     /**
 743      * Returns a string corresponding to the week-in-the-m
onth code.
 744      * <P>
 745      * Need to find a better approach.
 746      *
 747      * @param count  an integer code representing the week
-in-the-month.
 748      *
 749      * @return a string corresponding to the week-in-the-m
onth code.
 750      */
 751     public static String weekInMonthToString(final int cou
nt) {
 752
 753         switch (count) {
 754             case SerialDate.FIRST_WEEK_IN_MONTH : return "
First";
 755             case SerialDate.SECOND_WEEK_IN_MONTH : return
"Second";
 756             case SerialDate.THIRD_WEEK_IN_MONTH : return "
Third";
 757             case SerialDate.FOURTH_WEEK_IN_MONTH : return
"Fourth";
 758             case SerialDate.LAST_WEEK_IN_MONTH : return "L
ast";
 759             default :
 760                 return "SerialDate.weekInMonthToString():
invalid code.";
 761         }
 762
 763     }
 764
 765     /**
 766      * Returns a string representing the supplied 'relativ
e'.
 767      * <P>
 768      * Need to find a better approach.
 769      *
 770      * @param relative  a constant representing the 'relat
ive'.
```

```
 771      *
 772      * @return a string representing the supplied 'relativ
e'.
 773      */
 774     public static String relativeToString(final int relati
ve) {
 775
 776         switch (relative) {
 777             case SerialDate.PRECEDING : return "Preceding"
;
 778             case SerialDate.NEAREST : return "Nearest";
 779             case SerialDate.FOLLOWING : return "Following"
;
 780             default : return "ERROR : Relative To String";
 781         }
 782
 783     }
 784
 785     /**
 786      * Factory method that returns an instance of some con
crete subclass of
 787      * {@link SerialDate}.
 788      *
 789      * @param day  the day (1-31).
 790      * @param month  the month (1-12).
 791      * @param yyyy  the year (in the range 1900 to 9999).
 792      *
 793      * @return An instance of {@link SerialDate}.
 794      */
 795     public static SerialDate createInstance(final int day,
 final int month,
 796                                             final int yyyy
) {
 797         return new SpreadsheetDate(day, month, yyyy);
 798     }
 799
 800     /**
 801      * Factory method that returns an instance of some con
crete subclass of
 802      * {@link SerialDate}.
 803      *
 804      * @param serial  the serial number for the day (1 Jan
uary 1900 = 2).
 805      *
 806      * @return a instance of SerialDate.
```

```
 807      */
 808     public static SerialDate createInstance(final int seri
al) {
 809         return new SpreadsheetDate(serial);
 810     }
 811
 812     /**
 813      * Factory method that returns an instance of a subcla
ss of SerialDate.
 814      *
 815      * @param date  A Java date object.
 816      *
 817      * @return a instance of SerialDate.
 818      */
 819     public static SerialDate createInstance(final java.uti
l.Date date) {
 820
 821         final GregorianCalendar calendar = new GregorianCa
lendar();
 822         calendar.setTime(date);
 823         return new SpreadsheetDate(calendar.get(Calendar.D
ATE),
 824                                    calendar.get(Calendar.M
ONTH) + 1,
 825                                    calendar.get(Calendar.Y
EAR));
 826
 827     }
 828
 829     /**
 830      * Returns the serial number for the date, where 1 Jan
uary 1900 = 2 (this
 831      * corresponds, almost, to the numbering system used i
n Microsoft Excel for
 832      * Windows and Lotus 1-2-3).
 833      *
 834      * @return the serial number for the date.
 835      */
 836     public abstract int toSerial();
 837
 838     /**
 839      * Returns a java.util.Date.  Since java.util.Date has
 more precision than
 840      * SerialDate, we need to define a convention for the
'time of day'.
```

```
 841       *
 842       * @return this as <code>java.util.Date</code>.
 843       */
 844      public abstract java.util.Date toDate();
 845
 846      /**
 847       * Returns a description of the date.
 848       *
 849       * @return a description of the date.
 850       */
 851      public String getDescription() {
 852          return this.description;
 853      }
 854
 855      /**
 856       * Sets the description for the date.
 857       *
 858       * @param description  the new description for the dat
e.
 859       */
 860      public void setDescription(final String description) {
 861          this.description = description;
 862      }
 863
 864      /**
 865       * Converts the date to a string.
 866       *
 867       * @return  a string representation of the date.
 868       */
 869      public String toString() {
 870          return getDayOfMonth() + "-" + SerialDate.monthCod
eToString(getMonth())
 871                                 + "-" + getYYYY();
 872      }
 873
 874      /**
 875       * Returns the year (as


sume a valid range of 1900 to 9999).
 876       *
 877       * @return the year.
 878       */
 879      public abstract int getYYYY();
 880
```

```
 881     /**
 882      * Returns the month (January = 1, February = 2, March
 = 3).
 883      *
 884      * @return the month of the year.
 885      */
 886     public abstract int getMonth();
 887
 888     /**
 889      * Returns the day of the month.
 890      *
 891      * @return the day of the month.
 892      */
 893     public abstract int getDayOfMonth();
 894
 895     /**
 896      * Returns the day of the week.
 897      *
 898      * @return the day of the week.
 899      */
 900     public abstract int getDayOfWeek();
 901
 902     /**
 903      * Returns the difference (in days) between this date
and the specified
 904      * 'other' date.
 905      * <P>
 906      * The result is positive if this date is after the 'o
ther' date and
 907      * negative if it is before the 'other' date.
 908      *
 909      * @param other  the date being compared to.
 910      *
 911      * @return the difference between this and the other d
ate.
 912      */
 913     public abstract int compare(SerialDate other);
 914
 915     /**
 916      * Returns true if this SerialDate represents the same
 date as the
 917      * specified SerialDate.
 918      *
 919      * @param other  the date being compared to.
 920      *
```

```
 921      * @return <code>true</code> if this SerialDate repres
ents the same date as
 922      *          the specified SerialDate.
 923      */
 924     public abstract boolean isOn(SerialDate other);
 925
 926     /**
 927      * Returns true if this SerialDate represents an earli
er date compared to
 928      * the specified SerialDate.
 929      *
 930      * @param other  The date being compared to.
 931      *
 932      * @return <code>true</code> if this SerialDate repres
ents an earlier date
 933      *           compared to the specified SerialDate.
 934      */
 935     public abstract boolean isBefore(SerialDate other);
 936
 937     /**
 938      * Returns true if this SerialDate represents the same
 date as the
 939      * specified SerialDate.
 940      *
 941      * @param other  the date being compared to.
 942      *
 943      * @return <code>true<code> if this SerialDate represe
nts the same date
 944      *          as the specified SerialDate.
 945      */
 946     public abstract boolean isOnOrBefore(SerialDate other)
;
 947
 948     /**
 949      * Returns true if this SerialDate represents the same
 date as the
 950      * specified SerialDate.
 951      *
 952      * @param other  the date being compared to.
 953      *
 954      * @return <code>true</code> if this SerialDate repres
ents the same date
 955      *          as the specified SerialDate.
 956      */
 957     public abstract boolean isAfter(SerialDate other);
```

```
958
959     /**
960      * Returns true if this SerialDate represents the same date as the
961      * specified SerialDate.
962      *
963      * @param other  the date being compared to.
964      *
965      * @return <code>true</code> if this SerialDate represents the same date
966      *           as the specified SerialDate.
967      */
968     public abstract boolean isOnOrAfter(SerialDate other);
969
970     /**
971      * Returns <code>true</code> if this {@link SerialDate} is within the
972      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
973      * important.
974      *
975      * @param d1  a boundary date for the range.
976      * @param d2  the other boundary date for the range.
977      *
978      * @return A boolean.
979      */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983      * Returns <code>true</code> if this {@link SerialDate} is within the
984      * specified range (caller specifies whether or not the end-points are
985      * included).  The date order of d1 and d2 is not important.
986      *
987      * @param d1  a boundary date for the range.
988      * @param d2  the other boundary date for the range.
989      * @param include  a code that controls whether or not the start and end
990      *                 dates are included in the range.
991      *
992      * @return A boolean.
993      */
```

```
 994     public abstract boolean isInRange(SerialDate d1, Seria
lDate d2,
 995                                       int include);
 996
 997     /**
 998      * Returns the latest date that falls on the specified
 day-of-the-week and
 999      * is BEFORE this date.
1000      *
1001      * @param targetDOW  a code for the target day-of-the-
week.
1002      *
1003      * @return the latest date that falls on the specified
 day-of-the-week and
1004      *           is BEFORE this date.
1005      */
1006     public SerialDate getPreviousDayOfWeek(final int targe
tDOW) {
1007         return getPreviousDayOfWeek(targetDOW, this);
1008     }
1009
1010     /**
1011      * Returns the earliest date that falls on the specifi
ed day-of-the-week
1012      * and is AFTER this date.
1013      *
1014      * @param targetDOW  a code for the target day-of-the-
week.
1015      *
1016      * @return the earliest date that falls on the specifi
ed day-of-the-week
1017      *           and is AFTER this date.
1018      */
1019     public SerialDate getFollowingDayOfWeek(final int targ
etDOW) {
1020          return getFollowingDayOfWeek(targetDOW, this);
1021     }
1022
1023     /**
1024      * Returns the nearest date that falls on the specifie
d day-of-the-week.
1025      *
1026      * @param targetDOW  a code for the target day-of-the-
week.
1027      *
```

```
1028      * @return the nearest date that falls on the specifie
d day-of-the-week.
1029      */
1030     public SerialDate getNearestDayOfWeek(final int target
DOW) {
1031         return getNearestDayOfWeek(targetDOW, this);
1032     }
1033
1034 }
```

代码清单B-2 SerialDateTest.java

```
  1 /* ========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
  7  * Project Info:  http://www.jfree.org/jcommon/index.html
  8  *
  9  * This library is free software; you can redistribute it and/or modify it
 10  * under the terms of the GNU Lesser General Public License as published by
 11  * the Free Software Foundation; either version 2.1 of the License, or
 12  * (at your option) any later version.
 13  *
 14  * This library is distributed in the hope that it will be useful, but
 15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 17  * License for more details.
 18  *
 19  * You should have received a copy of the GNU Lesser General Public
 20  * License along with this library; if not, write to the Free Software
 21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 22  * USA.
 23  *
 24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 25  * in the United States and other countries.]
 26  *
 27  * --------------------
 28  * SerialDateTests.java
 29  * --------------------
 30  * (C) Copyright 2001-2005, by Object Refinery Limited.
 31  *
 32  * Original Author:  David Gilbert (for Object Refinery Limited);
 33  * Contributor(s):   -;
 34  *
```

```
 35  * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 36  *
 37  * Changes
 38  * -------
 39  * 15-Nov-2001 : Version 1 (DG);
 40  * 25-Jun-2002 : Removed unnecessary import (DG);
 41  * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 42  * 13-Mar-2003 : Added serialization test (DG);
 43  * 05-Jan-2005 : Added test for bug report 1096282 (DG);
 44  *
 45  */
 46
 47 package org.jfree.date.junit;
 48
 49 import java.io.ByteArrayInputStream;
 50 import java.io.ByteArrayOutputStream;
 51 import java.io.ObjectInput;
 52 import java.io.ObjectInputStream;
 53 import java.io.ObjectOutput;
 54 import java.io.ObjectOutputStream;
 55
 56 import junit.framework.Test;
 57 import junit.framework.TestCase;
 58 import junit.framework.TestSuite;
 59
 60 import org.jfree.date.MonthConstants;
 61 import org.jfree.date.SerialDate;
 62
 63 /**
 64  * Some JUnit tests for the {@link SerialDate} class.
 65  */
 66 public class SerialDateTests extends TestCase {
 67
 68     /** Date representing November 9. */
 69     private SerialDate nov9Y2001;
 70
 71     /**
 72      * Creates a new test case.
 73      *
 74      * @param name  the name.
 75      */
 76     public SerialDateTests(final String name) {
 77          super(name);
 78     }
 79
 80     /**
 81      * Returns a test suite for the JUnit test runner.
 82      *
 83      * @return The test suite.
 84      */
 85     public static Test suite() {
 86          return new TestSuite(SerialDateTests.class);
 87     }
 88
 89     /**
 90      * Problem set up.
 91      */
 92     protected void setUp() {
 93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER,
2001);
 94     }
```

```
 95
 96     /**
 97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
 98      */
 99     public void testAddMonthsTo9Nov2001() {
100          final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101          final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102          assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * A test case for a reported bug, now fixed.
107      */
108     public void testAddMonthsTo5Oct2003() {
109          final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCT
OBER,2003);
110          final SerialDate d2 = SerialDate.addMonths(2, d1);
111          assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER
, 2003));
112     }
113
114     /**
115      * A test case for a reported bug, now fixed.
116      */
117     public void testAddMonthsTo1Jan2003() {
118           final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JA
NUARY, 2003);
119          final SerialDate d2 = SerialDate.addMonths(0, d1);
120          assertEquals(d2, d1);
121     }
122
123     /**
124      * Monday preceding Friday 9 November 2001 should be 5 November.
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127          SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128              SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132
133     /**
134      * Monday following Friday 9 November 2001 should be 12 November.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137          SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138               SerialDate.MONDAY, this.nov9Y2001
139          );
140          assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142
143     /**
144      * Monday nearest Friday 9 November 2001 should be 12 November.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148              SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152
```

```
153     /**
154      * The Monday nearest to 22nd January 1970 falls on the 19th.
155      */
156     public void testMondayNearest22Jan1970() {
157          SerialDate jan22Y1970 = SerialDate.createInstance
             (22, MonthConstants.JANUARY, 1970);
158          SerialDate mondayNearest=SerialDate.getNearestDayOfWeek
             (SerialDate.MONDAY, jan22Y1970);
159          assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161
162     /**
163      * Problem that the conversion of days to strings returns the right result
.
         * Actually, this
164      * result depends on the Locale so this test needs to be modified.
165      */
166     public void testWeekdayCodeToString() {
167
168          final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDA
Y);
169          assertEquals("Saturday", test);
170
171     }
172
173     /**
174      * Test the conversion of a string to a weekday.  Note that this test will

           fail if the
175      * default locale doesn't use English weekday names...devise a better test
!
176      */
177     public void testStringToWeekday() {
178
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191      * Test the conversion of a string to a month.  Note that this test will
           fail if the
192      * default locale doesn't use English month names...devise a better test!
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
198
199         m = SerialDate.stringToMonthCode(" January ");
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
```

```
205     }
206
207     /**
208      * Tests the conversion of a month code to a string.
209      */
210     public void testMonthCodeToStringCode() {
211
212          final String test = SerialDate.monthCodeToString(MonthConstants.DECEM
BER);
213          assertEquals("December", test);
214
215     }
216
217     /**
218      * 1900 is not a leap year.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * 2000 is a leap year.
226      */
227     public void testIsLeapYear2000() {
228          assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * The number of leap years from 1900 up-to-and-including 1899 is 0.
233      */
234     public void testLeapYearCount1899() {
235          assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * The number of leap years from 1900 up-to-and-including 1903 is 0.
240      */
241     public void testLeapYearCount1903() {
242          assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * The number of leap years from 1900 up-to-and-including 1904 is 1.
247      */
248     public void testLeapYearCount1904() {
249          assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * The number of leap years from 1900 up-to-and-including 1999 is 24.
254      */
255     public void testLeapYearCount1999() {
256          assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * The number of leap years from 1900 up-to-and-including 2000 is 25.
261      */
262     public void testLeapYearCount2000() {
263          assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
```

```
265
266     /**
267      * Serialize an instance, restore it, and check for equality.
268      */
269     public void testSerialization() {
270
271          SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272          SerialDate d2 = null;
273
274         try {
275              ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276              ObjectOutput out = new ObjectOutputStream(buffer);
277              out.writeObject(d1);
278              out.close();
279
280              ObjectInput in = new ObjectInputStream
                                     (new ByteArrayInputStream(buffer.toByteArray
()));
281              d2 = (SerialDate) in.readObject();
282              in.close();
283         }
284         catch (Exception e) {
285              System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * A test for bug report 1096282 (now fixed).
293      */
294     public void test1096282() {
295          SerialDate d = SerialDate.createInstance(29, 2, 2004);
296          d = SerialDate.addYears(1, d);
297          SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298          assertTrue(d.isOn(expected));
299     }
300
301     /**
302      * Miscellaneous tests for the addMonths() method.
303      */
304     public void testAddMonths() {
305          SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
307          SerialDate d2 = SerialDate.addMonths(1, d1);
308          assertEquals(30, d2.getDayOfMonth());
309          assertEquals(6, d2.getMonth());
310          assertEquals(2004, d2.getYYYY());
311
312          SerialDate d3 = SerialDate.addMonths(2, d1);
313          assertEquals(31, d3.getDayOfMonth());
314          assertEquals(7, d3.getMonth());
315          assertEquals(2004, d3.getYYYY());
316
317          SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318          assertEquals(30, d4.getDayOfMonth());
319          assertEquals(7, d4.getMonth());
320          assertEquals(2004, d4.getYYYY());
321     }
322 }
```

代码清单B-3 MonthConstants.java

```
 1 /* ========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info: http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * -------------------
28  * MonthConstants.java
29  * -------------------
30  * (C) Copyright 2002, 2003, by Object Refinery Limited.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
33  * Contributor(s):   -;
34  *
35  * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36  *
37  * Changes
38  * -------
39  * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
40  *
41  */
42
43 package org.jfree.date;
44
45 /**
46  * Useful constants for months.  Note that these are NOT equivalent to the
47  * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
48  * <P>
49  * Used by the SerialDate and RegularTimePeriod classes.
50  *
51  * @author David Gilbert
52  */
53 public interface MonthConstants {
54
55      /** Constant for January. */
56      public static final int JANUARY = 1;
57
58      /** Constant for February. */
59      public static final int FEBRUARY = 2;
60
61      /** Constant for March. */
62      public static final int MARCH = 3;
63
64      /** Constant for April. */
65      public static final int APRIL = 4;
66
67     /** Constant for May. */
68     public static final int MAY = 5;
69
70     /** Constant for June. */
71     public static final int JUNE = 6;
```

```
72
73     /** Constant for July. */
74     public static final int JULY = 7;
75
76     /** Constant for August. */
77     public static final int AUGUST = 8;
78
79     /** Constant for September. */
80     public static final int SEPTEMBER = 9;
81
82     /** Constant for October. */
83     public static final int OCTOBER = 10;
84
85     /** Constant for November. */
86     public static final int NOVEMBER = 11;
87
88     /** Constant for December. */
89     public static final int DECEMBER = 12;
90
91 }
```

代码清单B-4 BobsSerialDateTest.java

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo    assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 //    assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 //    assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //    assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 //    assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 //    assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //    assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 //    assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
37 //    assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 //    assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 //    assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 //    assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 //    assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 //    assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 //    assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 //    assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
```

```
 55 //    assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
 56 //    assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
 57 //    assertEquals(SATURDAY, stringToWeekdayCode("sat"));
 58
 59     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
 60     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
 61 //    assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
 62 //    assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
 63 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
 64   }
 65
 66   public void testWeekdayCodeToString() throws Exception {
 67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
 68     assertEquals("Monday", weekdayCodeToString(MONDAY));
 69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
 70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
 71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
 72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
 73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
 74   }
 75
 76   public void testIsValidMonthCode() throws Exception {
 77     for (int i = 1; i <= 12; i++)
 78       assertTrue(isValidMonthCode(i));
 79     assertFalse(isValidMonthCode(0));
 80     assertFalse(isValidMonthCode(13));
 81   }
 82
 83   public void testMonthToQuarter() throws Exception {
 84     assertEquals(1, monthCodeToQuarter(JANUARY));
 85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
 86     assertEquals(1, monthCodeToQuarter(MARCH));
 87     assertEquals(2, monthCodeToQuarter(APRIL));
 88     assertEquals(2, monthCodeToQuarter(MAY));
 89     assertEquals(2, monthCodeToQuarter(JUNE));
 90     assertEquals(3, monthCodeToQuarter(JULY));
 91     assertEquals(3, monthCodeToQuarter(AUGUST));
 92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
 93     assertEquals(4, monthCodeToQuarter(OCTOBER));
 94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
 95     assertEquals(4, monthCodeToQuarter(DECEMBER));
 96
 97     try {
 98        monthCodeToQuarter(-1);
 99        fail("Invalid Month Code should throw exception");
100     }  catch (IllegalArgumentException e) {
101     }
102   }
103
104   public void testMonthCodeToString() throws Exception {
105      assertEquals("January", monthCodeToString(JANUARY));
106      assertEquals("February", monthCodeToString(FEBRUARY));
107      assertEquals("March", monthCodeToString(MARCH));
108      assertEquals("April", monthCodeToString(APRIL));
109      assertEquals("May", monthCodeToString(MAY));
110      assertEquals("June", monthCodeToString(JUNE));
111      assertEquals("July", monthCodeToString(JULY));
112      assertEquals("August", monthCodeToString(AUGUST));
113      assertEquals("September", monthCodeToString(SEPTEMBER));
114      assertEquals("October", monthCodeToString(OCTOBER));
115      assertEquals("November", monthCodeToString(NOVEMBER));
116      assertEquals("December", monthCodeToString(DECEMBER));
117
118      assertEquals("Jan", monthCodeToString(JANUARY, true));
119      assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120      assertEquals("Mar", monthCodeToString(MARCH, true));
121      assertEquals("Apr", monthCodeToString(APRIL, true));
122      assertEquals("May", monthCodeToString(MAY, true));
123      assertEquals("Jun", monthCodeToString(JUNE, true));
124      assertEquals("Jul", monthCodeToString(JULY, true));
125      assertEquals("Aug", monthCodeToString(AUGUST, true));
126      assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127      assertEquals("Oct", monthCodeToString(OCTOBER, true));
128      assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129      assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132        monthCodeToString(-1);
133        fail("Invalid month code should throw exception");
134     }  catch (IllegalArgumentException e) {
135     }
136
137   }
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
```

```
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo    assertEquals(-1, stringToMonthCode("0"));
154 //    assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 //    assertEquals(1,stringToMonthCode("jan"));
164 //    assertEquals(2,stringToMonthCode("feb"));
165 //    assertEquals(3,stringToMonthCode("mar"));
166 //    assertEquals(4,stringToMonthCode("apr"));
167 //    assertEquals(5,stringToMonthCode("may"));
168 //    assertEquals(6,stringToMonthCode("jun"));
169 //    assertEquals(7,stringToMonthCode("jul"));
170 //    assertEquals(8,stringToMonthCode("aug"));
171 //    assertEquals(9,stringToMonthCode("sep"));
172 //    assertEquals(10,stringToMonthCode("oct"));
173 //    assertEquals(11,stringToMonthCode("nov"));
174 //    assertEquals(12,stringToMonthCode("dec"));
175
176 //    assertEquals(1,stringToMonthCode("JAN"));
177 //    assertEquals(2,stringToMonthCode("FEB"));
178 //    assertEquals(3,stringToMonthCode("MAR"));
179 //    assertEquals(4,stringToMonthCode("APR"));
180 //    assertEquals(5,stringToMonthCode("MAY"));
181 //    assertEquals(6,stringToMonthCode("JUN"));
182 //    assertEquals(7,stringToMonthCode("JUL"));
183 //    assertEquals(8,stringToMonthCode("AUG"));
184 //    assertEquals(9,stringToMonthCode("SEP"));
185 //    assertEquals(10,stringToMonthCode("OCT"));
186 //    assertEquals(11,stringToMonthCode("NOV"));
187 //    assertEquals(12,stringToMonthCode("DEC"));
188
189 //    assertEquals(1,stringToMonthCode("january"));
190 //    assertEquals(2,stringToMonthCode("february"));
191 //    assertEquals(3,stringToMonthCode("march"));
192 //    assertEquals(4,stringToMonthCode("april"));
193 //    assertEquals(5,stringToMonthCode("may"));
194 //    assertEquals(6,stringToMonthCode("june"));
195 //    assertEquals(7,stringToMonthCode("july"));
196 //    assertEquals(8,stringToMonthCode("august"));
197 //    assertEquals(9,stringToMonthCode("september"));
198 //    assertEquals(10,stringToMonthCode("october"));
199 //    assertEquals(11,stringToMonthCode("november"));
200 //    assertEquals(12,stringToMonthCode("december"));
201
202 //    assertEquals(1,stringToMonthCode("JANUARY"));
203 //    assertEquals(2,stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3,stringToMonthCode("MAR"));
205 //    assertEquals(4,stringToMonthCode("APRIL"));
206 //    assertEquals(5,stringToMonthCode("MAY"));
207 //    assertEquals(6,stringToMonthCode("JUNE"));
208 //    assertEquals(7,stringToMonthCode("JULY"));
209 //    assertEquals(8,stringToMonthCode("AUGUST"));
210 //    assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10,stringToMonthCode("OCTOBER"));
212 //    assertEquals(11,stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
```

```
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
SpreadsheetDate(day, month, year);}
281
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY,
         d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY,
         d(1, MARCH, 2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY,
         d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY,
```

```
          d(5, JANUARY, 2005)));
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Invalid day of week code should throw exception");
313     } catch (IllegalArgumentException e) {
314     }
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY,
          d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY,
        d(26, DECEMBER, 2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY,
        d(28, FEBRUARY, 2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18


, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo    assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY,
                d(16, APRIL, 2006)));
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY,
           d(16, APRIL, 2006)));
347 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY,
           d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY
          d(16, APRIL, 2006)));
355 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
          d(17, APRIL, 2006)));
356 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
          d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
        d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
        d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
        d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY,
        d(22, APRIL, 2006)));
361
362 //    assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
          d(16, APRIL, 2006)));
363 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
          d(17, APRIL, 2006)));
364 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
          d(18, APRIL, 2006)));
365 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
          d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
        d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
        d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY,
        d(22, APRIL, 2006)));
369
370 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
```

```
371 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(16, APRIL, 2006)));
379 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(17, APRIL, 2006)));
380 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(18, APRIL, 2006)));
381 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(19, APRIL, 2006)));
382 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(20, APRIL, 2006)));
383 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
        d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY,
      d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Invalid day of week code should throw exception");
389     } catch (IllegalArgumentException e) {
390     }
391   }
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth
      (d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo    try {
418 //      weekInMonthToString(-1);
419 //      fail("Invalid week code should throw exception");
420 //    } catch (IllegalArgumentException e) {
421 //    }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo    try {
430 //      relativeToString(-1000);
431 //      fail("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434   }
435
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
```

```
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
                  createInstance(new GregorianCalendar(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),
    createInstance(new GregorianCalendar(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

代码清单B-5 SpreadsheetDate.java

```
  1 /* ========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
  7  * Project Info:  http://www.jfree.org/jcommon/index.html
  8  *
  9  * This library is free software; you can redistribute it and/or modify it
 10  * under the terms of the GNU Lesser General Public License as published by
 11  * the Free Software Foundation; either version 2.1 of the License, or
 12  * (at your option) any later version.
 13  *
 14  * This library is distributed in the hope that it will be useful, but
 15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 17  * License for more details.
 18  *
 19  * You should have received a copy of the GNU Lesser General Public
 20  * License along with this library; if not, write to the Free Software
 21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 22  * USA.
 23  *
 24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 25  * in the United States and other countries.]
 26  *
 27  * --------------------
 28  * SpreadsheetDate.java
 29  * --------------------
 30  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 31  *
 32  * Original Author:  David Gilbert (for Object Refinery Limited);
 33  * Contributor(s):   -;
 34  *
 35  * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
 36  *
 37  * Changes
 38  * -------
 39  * 11-Oct-2001 : Version 1 (DG);
 40  * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
 41  * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
 42  *               Fixed a bug in calculating day, month and year from serial
 43  *               number (DG);
 44  * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
 45  *               month and year.  Thanks to Trevor Hills for the report (DG);
 46  * 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
 47  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 48  * 13-Mar-2003 : Implemented Serializable (DG);
 49  * 04-Sep-2003 : Completed isInRange() methods (DG);
 50  * 05-Sep-2003 : Implemented Comparable (DG);
 51  * 21-Oct-2003 : Added hashCode() method (DG);
 52  *
 53  */
 54
 55 package org.jfree.date;
 56
 57 import java.util.Calendar;
 58 import java.util.Date;
 59
 60 /**
 61  * Represents a date using an integer, in a similar fashion to the
 62  * implementation in Microsoft Excel.  The range of dates supported is
 63  * 1-Jan-1900 to 31-Dec-9999.
 64  * <P>
 65  * Be aware that there is a deliberate bug in Excel that recognises the year
 66  * 1900 as a leap year when in fact it is not a leap year. You can find more
 67  * information on the Microsoft website in article Q181370:
```

```
 68  * <P>
 69  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 70  * <P>
 71  * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 72  * convention 1-Jan-1900 = 2.
 73  * The result is that the day number in this class will be different to the
 74  * Excel figure for January and February 1900...but then Excel adds in an extra
 75  * day (29-Feb-1900 which does not actually exist!) and from that point forward
 76  * the day numbers will match.
 77  *
 78  * @author David Gilbert
 79  */
 80 public class SpreadsheetDate extends SerialDate {
 81 
 82     /** For serialization. */
 83     private static final long serialVersionUID = -2039586705374454461L;
 84 
 85     /**
 86      * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
 87      * 2958465).
 88      */
 89     private int serial;
 90 
 91     /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
 92     private int day;
 93 
 94     /** The month of the year (1 to 12). */
 95     private int month;
 96 
 97     /** The year (1900 to 9999). */
 98     private int year;
 99 
100     /** An optional description for the date. */
101     private String description;
102 
103     /**
104      * Creates a new date instance.
105      *
106      * @param day  the day (in the range 1 to 28/29/30/31).
107      * @param month  the month (in the range 1 to 12).
108      * @param year  the year (in the range 1900 to 9999).
109      */
110     public SpreadsheetDate(final int day, final int month, final int year) {
111 
112         if ((year >= 1900) && (year <= 9999)) {
113             this.year = year;
114         }
115         else {
116             throw new IllegalArgumentException(
117                  "The 'year' argument must be in range 1900 to 9999."
118             );
119         }
120 
121         if ((month >= MonthConstants.JANUARY)
122                 && (month <= MonthConstants.DECEMBER)) {
123             this.month = month;
124         }
125         else {
126             throw new IllegalArgumentException(
127                 "The 'month' argument must be in the range 1 to 12."
128             );
129         }
130 
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
136         }
137 
138         // the serial number needs to be synchronised with the day-month-year...
139         this.serial = calcSerial(day, month, year);
140 
141         this.description = null;
142 
143     }
144 
145     /**
146      * Standard constructor - creates a new date object representing the
147      * specified day number (which should be in the range 2 to 2958465.
148      *
149      * @param serial  the serial number for the day (range: 2 to 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152 
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
157             throw new IllegalArgumentException(
158                  "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160 
```

```
161         // the day-month-year needs to be synchronised with the serial number...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Returns the description that is attached to the date.  It is not
168      * required that a date have a description, but for some applications it
169      * is useful.
170      *
171      * @return The description that is attached to the date.
172      */
173     public String getDescription() {
174          return this.description;
175     }
176
177     /**
178      * Sets the description for the date.
179      *
180      * @param description  the description for this date (<code>null</code>
181      *                         permitted).
182      */
183     public void setDescription(final String description) {
184          this.description = description;
185     }
186
187     /**
188      * Returns the serial number for the date, where 1 January 1900 = 2
189      * (this corresponds, almost, to the numbering system used in Microsoft
190      * Excel for Windows and Lotus 1-2-3).
191      *
192      * @return The serial number of this date.
193      */
194     public int toSerial() {
195          return this.serial;
196     }
197
198     /**
199      * Returns a <code>java.util.Date</code> equivalent to this date.
200      *
201      * @return The date.
202      */
203     public Date toDate() {
204          final Calendar calendar = Calendar.getInstance();
205          calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206          return calendar.getTime();
207     }
208
209     /**
210      * Returns the year (assume a valid range of 1900 to 9999).
211      *
212      * @return The year.
213      */
214     public int getYYYY() {
215          return this.year;
216     }
217
218     /**
219      * Returns the month (January = 1, February = 2, March = 3).
220      *
221      * @return The month of the year.
222      */
223     public int getMonth() {
224          return this.month;
225     }
226
227     /**
228      * Returns the day of the month.
229      *
230      * @return The day of the month.
231      */
232     public int getDayOfMonth() {
233          return this.day;
234     }
235
236     /**
237      * Returns a code representing the day of the week.
238      * <P>
239      * The codes are defined in the {@link SerialDate} class as:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242      * <code>SATURDAY</code>.
243      *
244      * @return A code representing the day of the week.
245      */
246     public int getDayOfWeek() {
247          return (this.serial + 6) % 7 + 1;
248     }
249
250     /**
251      * Tests the equality of this date with an arbitrary object.
252      * <P>
253      * This method will return true ONLY if the object is an instance of the
254      * {@link SerialDate} base class, and it represents the same day as this
255      * {@link SpreadsheetDate}.
```

```
256      *
257      * @param object  the object to compare (<code>null</code> permitted).
258      *
259      * @return A boolean.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264              final SerialDate s = (SerialDate) object;
265              return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270
271     }
272
273     /**
274      * Returns a hash code for this object instance.
275      *
276      * @return A hash code.
277      */
278     public int hashCode() {
279          return toSerial();
280     }
281
282     /**
283      * Returns the difference (in days) between this date and the specified
284      * 'other' date.
285      *
286      * @param other  the date being compared to.
287      *
288      * @return The difference (in days) between this date and the specified
289      *          'other' date.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implements the method required by the Comparable interface.
297      *
298      * @param other  the other object (usually another SerialDate).
299      *
300      * @return A negative integer, zero, or a positive integer as this object
301      *          is less than, equal to, or greater than the specified object.
302      */
303     public int compareTo(final Object other) {
304          return compare((SerialDate) other);
305     }
306
307     /**
308      * Returns true if this SerialDate represents the same date as the
309      * specified SerialDate.
310      *
311      * @param other  the date being compared to.
312      *
313      * @return <code>true</code> if this SerialDate represents the same date as
314      *          the specified SerialDate.
315      */
316     public boolean isOn(final SerialDate other) {
317          return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Returns true if this SerialDate represents an earlier date compared to
322      * the specified SerialDate.
323      *
324      * @param other  the date being compared to.
325      *
326      * @return <code>true</code> if this SerialDate represents an earlier date
327      *          compared to the specified SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330          return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Returns true if this SerialDate represents the same date as the
335      * specified SerialDate.
336      *
337      * @param other  the date being compared to.
338      *
339      * @return <code>true</code> if this SerialDate represents the same date
340      *          as the specified SerialDate.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Returns true if this SerialDate represents the same date as the
348      * specified SerialDate.
349      *
350      * @param other  the date being compared to.
```

```
351         *
352         * @return <code>true</code> if this SerialDate represents the same date
353         *          as the specified SerialDate.
354         */
355        public boolean isAfter(final SerialDate other) {
356             return (this.serial > other.toSerial());
357        }
358
359        /**
360         * Returns true if this SerialDate represents the same date as the
361         * specified SerialDate.
362         *
363         * @param other  the date being compared to.
364         *
365         * @return <code>true</code> if this SerialDate represents the same date as
366         *          the specified SerialDate.
367         */
368        public boolean isOnOrAfter(final SerialDate other) {
369             return (this.serial >= other.toSerial());
370        }
371
372        /**
373         * Returns <code>true</code> if this {@link SerialDate} is within the
374         * specified range (INCLUSIVE).  The date order of d1 and d2 is not
375         * important.
376         *
377         * @param d1  a boundary date for the range.
378         * @param d2  the other boundary date for the range.
379         *
380         * @return A boolean.
381         */
382        public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383             return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384        }
385
386        /**
387         * Returns true if this SerialDate is within the specified range (caller
388         * specifies whether or not the end-points are included).  The order of d1
389         * and d2 is not important.
390         *
391         * @param d1  one boundary date for the range.
392         * @param d2  a second boundary date for the range.
393         * @param include  a code that controls whether or not the start and end
394         *                 dates are included in the range.
395         *
396         * @return <code>true</code> if this SerialDate is within the specified
397         *        range.
398         */
399        public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                                 final int include) {
401            final int s1 = d1.toSerial();
402            final int s2 = d2.toSerial();
403            final int start = Math.min(s1, s2);
404            final int end = Math.max(s1, s2);
405
406            final int s = toSerial();
407            if (include == SerialDate.INCLUDE_BOTH) {
408                return (s >= start && s <= end);
409            }
410            else if (include == SerialDate.INCLUDE_FIRST) {
411                return (s >= start && s < end);
412            }
413            else if (include == SerialDate.INCLUDE_SECOND) {
414                return (s > start && s <= end);
415            }
416            else {
417                return (s > start && s < end);
418            }
419        }
420
421        /**
422         * Calculate the serial number from the day, month and year.
423         * <P>
424         * 1-Jan-1900 = 2.
425         *
426         * @param d  the day.
427         * @param m  the month.
428         * @param y  the year.
429         *
430         * @return the serial number from the day, month and year.
431         */
432        private int calcSerial(final int d, final int m, final int y) {
433            final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434            int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435            if (m > MonthConstants.FEBRUARY) {
436                if (SerialDate.isLeapYear(y)) {
437                     mm = mm + 1;
438                }
439            }
440            final int dd = d;
441            return yy + mm + dd + 1;
442        }
443
444        /**
445         * Calculate the day, month and year from the serial number.
446         */
```

```
447     private void calcDayMonthYear() {
448
449         // get the year from the serial date
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // overestimated because we ignored leap days
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // underestimated because we overestimated years
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // get the month from the serial date
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
489         // what's left is d(+1);
490         this.day = this.serial - ss2
491                    - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

代码清单B-6 RelativeDayOfWeekRule.java

```
 1 /* ========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * --------------------------
28  * RelativeDayOfWeekRule.java
29  * --------------------------
30  * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
33  * Contributor(s):   -;
```

```
 34  *
 35  * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 36  *
 37  * Changes (from 26-Oct-2001)
 38  * --------------------------
 39  * 26-Oct-2001 : Changed package to com.jrefinery.date.*;
 40  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 41  *
 42  */
 43
 44 package org.jfree.date;
 45
 46 /**
 47  * An annual date rule that returns a date for each year based on (a) a
 48  * reference rule; (b) a day of the week; and (c) a selection parameter
 49  * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
 50  * <P>
 51  * For example, Good Friday can be specified as 'the Friday PRECEDING Easter
 52  * Sunday'.
 53  *
 54  * @author David Gilbert
 55  */
 56 public class RelativeDayOfWeekRule extends AnnualDateRule {
 57
 58     /** A reference to the annual date rule on which this rule is based. */
 59     private AnnualDateRule subrule;
 60
 61     /**
 62      * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
 63      */
 64     private int dayOfWeek;
 65
 66     /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */
 67     private int relative;
 68
 69     /**
 70      * Default constructor - builds a rule for the Monday following 1 January.
 71      */
 72     public RelativeDayOfWeekRule() {
 73          this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
 74     }
 75
 76     /**
 77      * Standard constructor - builds rule based on the supplied sub-rule.
 78      *
 79      * @param subrule  the rule that determines the reference date.
 80      * @param dayOfWeek  the day-of-the-week relative to the reference date.
 81      * @param relative  indicates *which* day-of-the-week (preceding, nearest
 82      *                  or following).
 83      */
 84     public RelativeDayOfWeekRule(final AnnualDateRule subrule,
 85             final int dayOfWeek, final int relative) {
 86          this.subrule = subrule;
 87          this.dayOfWeek = dayOfWeek;
 88          this.relative = relative;
 89     }
 90
 91     /**
 92      * Returns the sub-rule (also called the reference rule).
 93      *
 94      * @return The annual date rule that determines the reference date for this
 95      *         rule.
 96      */
 97     public AnnualDateRule getSubrule() {
 98          return this.subrule;
 99     }
100
101     /**
102      * Sets the sub-rule.
103      *
104      * @param subrule  the annual date rule that determines the reference date
105      *                 for this rule.
106      */
107     public void setSubrule(final AnnualDateRule subrule) {
108          this.subrule = subrule;
109     }
110
111     /**
112      * Returns the day-of-the-week for this rule.
113      *
114      * @return the day-of-the-week for this rule.
115      */
116     public int getDayOfWeek() {
117          return this.dayOfWeek;
118     }
119
120     /**
121      * Sets the day-of-the-week for this rule.
122      *
123      * @param dayOfWeek  the day-of-the-week (SerialDate.MONDAY,
124      *                   SerialDate.TUESDAY, and so on).
125      */
126     public void setDayOfWeek(final int dayOfWeek) {
127         this.dayOfWeek = dayOfWeek;
128     }
```

```
129
130     /**
131      * Returns the 'relative' attribute, that determines *which*
132      * day-of-the-week we are interested in (SerialDate.PRECEDING,
133      * SerialDate.NEAREST or SerialDate.FOLLOWING).
134      *
135      * @return The 'relative' attribute.
136      */
137     public int getRelative() {
138          return this.relative;
139     }
140
141     /**
142      * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  determines *which* day-of-the-week is selected by this
146      *                  rule.
147      */
148     public void setRelative(final int relative) {
149          this.relative = relative;
150     }
151
152     /**
153      * Creates a clone of this rule.
154      *
155      * @return a clone of this rule.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160          final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162          duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163          return duplicate;
164     }
165
166     /**
167      * Returns the date generated by this rule, for the specified year.
168      *
169      * @param year  the year (1900 <= year <= 9999).
170      *
171      * @return The date generated by the rule for the given year (possibly
172      *         <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                       result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                                 base);
192                       break;
193                 case(SerialDate.NEAREST):
194                       result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                       break;
197                 case(SerialDate.FOLLOWING):
198                       result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                                 base);
200                       break;
201                 default:
202                       break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

代码清单B-7 DayDate.java（最终版本）

```
  1 /* ========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ========================================================================
  4  *
```

```
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
...
 36  */
 37 package org.jfree.date;
 38
 39 import java.io.Serializable;
 40 import java.util.*;
 41
 42 /**
 43  * An abstract class that represents immutable dates with a precision of
 44  * one day.  The implementation will map each date to an integer that
 45  * represents an ordinal number of days from some fixed origin.
 46  *
 47  * Why not just use java.util.Date?  We will, when it makes sense.  At times,
 48  * java.util.Date can be *too* precise - it represents an instant in time,
 49  * accurate to 1/1000th of a second (with the date itself depending on the
 50  * time-zone).  Sometimes we just want to represent a particular day (e.g. 21
 51  * January 2015) without concerning ourselves about the time of day, or the
 52  * time-zone, or anything else.  That's what we've defined DayDate for.
 53  *
 54  * Use DayDateFactory.makeDate to create an instance.
 55  *
 56  * @author David Gilbert
 57  * @author Robert C. Martin did a lot of refactoring.
 58  */
 59
 60 public abstract class DayDate implements Comparable, Serializable {
 61   public abstract int getOrdinalDay();
 62   public abstract int getYear();
 63   public abstract Month getMonth();
 64   public abstract int getDayOfMonth();
 65
 66   protected abstract Day getDayOfWeekForOrdinalZero();
 67
 68   public DayDate plusDays(int days) {
 69     return DayDateFactory.makeDate(getOrdinalDay() + days);
 70   }
 71
 72   public DayDate plusMonths(int months) {
 73     int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
 74     int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
 75     int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
 76     int resultYear = resultMonthAndYearAsOrdinal / 12;
 77     int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
 78     Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
 79     int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
 80     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
 81   }
 82
 83   public DayDate plusYears(int years) {
 84     int resultYear = getYear() + years;
 85     int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
 86     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
 87   }
 88
 89   private int correctLastDayOfMonth(int day, Month month, int year) {
 90     int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
 91     if (day > lastDayOfMonth)
 92         day = lastDayOfMonth;
 93     return day;
 94   }
 95
 96   public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
 97     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
 98     if (offsetToTarget >= 0)
 99       offsetToTarget -= 7;
100     return plusDays(offsetToTarget);
101   }
102
103   public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105     if (offsetToTarget <= 0)
106       offsetToTarget += 7;
107     return plusDays(offsetToTarget);
108   }
109
110   public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111     int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112     int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113     int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115     if (offsetToFutureTarget > 3)
116       return plusDays(offsetToPreviousTarget);
117     else
118       return plusDays(offsetToFutureTarget);
119   }
120
121   public DayDate getEndOfMonth() {
122     Month month = getMonth();
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126   }
127
128   public Date toDate() {
```

```
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133   }
134
135   public String toString() {
136     return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137   }
138
139   public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144   }
145
146   public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148   }
149
150   public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152   }
153
154   public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

代码清单B-8 Month.java（最终版本）

```
  1 package org.jfree.date;
  2
  3 import java.text.DateFormatSymbols;
  4
  5 public enum Month {
  6   JANUARY(1), FEBRUARY(2), MARCH(3),
  7   APRIL(4),   MAY(5),      JUNE(6),
  8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
  9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
 10   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 11   private static final int[] LAST_DAY_OF_MONTH =
 12     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
 13
 14   private int index;
 15
 16   Month(int index) {
 17     this.index = index;
 18   }
 19
 20   public static Month fromInt(int monthIndex) {
 21     for (Month m : Month.values()) {
 22       if (m.index == monthIndex)
 23         return m;
 24     }
 25     throw new IllegalArgumentException("Invalid month index " + monthIndex);
 26   }
 27
 28   public int lastDay() {
 29     return LAST_DAY_OF_MONTH[index];
 30   }
 31
 32   public int quarter() {
 33     return 1 + (index - 1) / 3;
 34   }
 35
```

```
36   public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38   }
39
40   public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42   }
43
44   public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47       if (m.matches(s))
48         return m;
49
50     try {
51       return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54     throw new IllegalArgumentException("Invalid month " + s);
55   }
56
57   private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59            s.equalsIgnoreCase(toShortString());
60   }
61
62   public int toInt() {
63     return index;
64   }
65 }
```

代码清单B-9 Day.java（最终版本）

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format("Illegal day index: %d.", index));
28   }
29
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32       dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34       dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44       String.format("%s is not a valid weekday string", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

代码清单B-10 DateInterval.java （最终版本）

```
 1 package org.jfree.date;
 2
 3 public enum DateInterval {
 4   OPEN {
 5     public boolean isIn(int d, int left, int right) {
 6       return d > left && d < right;
 7     }
 8   },
 9   CLOSED_LEFT {
10     public boolean isIn(int d, int left, int right) {
11       return d >= left && d < right;
12     }
13   },
14   CLOSED_RIGHT {
15     public boolean isIn(int d, int left, int right) {
16       return d > left && d <= right;
17     }
18   },
19   CLOSED {
20     public boolean isIn(int d, int left, int right) {
21       return d >= left && d <= right;
22     }
23   };
24
25   public abstract boolean isIn(int d, int left, int right);
26 }
```

代码清单B-11 WeekInMonth.java （最终版本）

```
 1 package org.jfree.date;
 2
 3 public enum WeekInMonth {
 4   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
 5   private final int index;
 6
 7   WeekInMonth(int index) {
 8     this.index = index;
 9   }
10
11   public int toInt() {
12     return index;
13   }
14 }
```

代码清单B-12 WeekdayRange.java （最终版本）

```
 1 package org.jfree.date;
 2
 3 public enum WeekdayRange {
 4   LAST, NEAREST, NEXT
 5 }
```

代码清单B-13 DateUtil.java （最终版本）

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public class DateUtil {
 6   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 7
 8   public static String[] getMonthNames() {
 9     return dateFormatSymbols.getMonths();
10   }
11
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
```

```
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

代码清单B-14 DayDateFactory.java （最终版本）

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
10   protected abstract DayDate _makeDate(int day, Month month, int year);
11   protected abstract DayDate _makeDate(int day, int month, int year);
12   protected abstract DayDate _makeDate(java.util.Date date);
13   protected abstract int _getMinimumYear();
14   protected abstract int _getMaximumYear();
15
16   public static DayDate makeDate(int ordinal) {
17     return factory._makeDate(ordinal);
18   }
19
20   public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22   }
23
24   public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26   }
27
28   public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30   }
31
32   public static int getMinimumYear() {
33     return factory._getMinimumYear();
34   }
35
36   public static int getMaximumYear() {
37     return factory._getMaximumYear();
38   }
39 }
```

代码清单B-15 SpreadsheetDateFactory.java（最终版本）

```
 1 package org.jfree.date;
 2
 3 import java.util.*;
 4
 5 public class SpreadsheetDateFactory extends DayDateFactory {
 6   public DayDate _makeDate(int ordinal) {
 7     return new SpreadsheetDate(ordinal);
 8   }
 9
10   public DayDate _makeDate(int day, Month month, int year) {
11     return new SpreadsheetDate(day, month, year);
12   }
13
14   public DayDate _makeDate(int day, int month, int year) {
15     return new SpreadsheetDate(day, month, year);
16   }
17
18   public DayDate _makeDate(Date date) {
```

```
19     final GregorianCalendar calendar = new GregorianCalendar();
20     calendar.setTime(date);
21     return new SpreadsheetDate(
22       calendar.get(Calendar.DATE),
23       Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24       calendar.get(Calendar.YEAR));
25   }
26
27   protected int _getMinimumYear() {
28     return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29   }
30
31   protected int _getMaximumYear() {
32     return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33   }
34 }
```

代码清单B-16 SpreadsheetDate.java（最终版本）

```
  1 /* ========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
..
 52  *
 53  */
 54
 55 package org.jfree.date;
 56
 57 import static org.jfree.date.Month.FEBRUARY;
 58
 59 import java.util.*;
 60
 61 /**
 62  * Represents a date using an integer, in a similar fashion to the
 63  * implementation in Microsoft Excel.  The range of dates supported is
 64  * 1-Jan-1900 to 31-Dec-9999.
 65  * <p/>
 66  * Be aware that there is a deliberate bug in Excel that recognises the year
 67  * 1900 as a leap year when in fact it is not a leap year. You can find more
 68  * information on the Microsoft website in article Q181370:
 69  * <p/>
 70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 71  * <p/>
 72  * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 73  * convention 1-Jan-1900 = 2.
 74  * The result is that the day number in this class will be different to the
 75  * Excel figure for January and February 1900...but then Excel adds in an extra
 76  * day (29-Feb-1900 which does not actually exist!) and from that point forward
 77  * the day numbers will match.
 78  *
 79  * @author David Gilbert
 80  */
 81 public class SpreadsheetDate extends DayDate {
 82   public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
 83   public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
 84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
 85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
 86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
 88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
 90
 91   private int ordinalDay;
 92   private int day;
 93   private Month month;
 94   private int year;
 95
 96   public SpreadsheetDate(int day, Month month, int year) {
 97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
 98       throw new IllegalArgumentException(
 99         "The 'year' argument must be in range " +
100         MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101     if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102       throw new IllegalArgumentException("Invalid 'day' argument.");
103
104     this.year = year;
105     this.month = month;
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108   }
109
110   public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
```

```
112   }
113
114   public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116       throw new IllegalArgumentException(
117         "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121   }
122
123   public int getOrdinalDay() {
124     return ordinalDay;
125   }
126
127   public int getYear() {
128     return year;
129   }
130
131   public Month getMonth() {
132     return month;
133   }
134
135   public int getDayOfMonth() {
136     return day;
137   }
138
139   protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141   public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143       return false;
144
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147   }
148
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186   }
187
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199
200     return aYear - 1;
201   }
202
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205   }
206
```

```
207   public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213
214   }
215 }
```

结束语

2005年，在参加于丹佛举行的敏捷大会时，Elisabeth Hedrickson [1] 递给我一条类似Lance Armstrong热销的那种绿色腕带。这条腕带上面写着“沉迷测试”（Test Obsessed）的字样。我高兴地戴上，并自豪地一直系着。自从1999年从Kent Beck那儿学到TDD以来，我的确迷上了测试驱动开发。

不过跟着就发生了些奇事。我发现自己无法取下腕带。不仅是因为腕带很紧，而且那也是条精神上的紧箍咒。那腕带就是我职业道德的宣告，也是我承诺尽己所能写出最好代码的提示。取下它，仿佛就是违背了这些宣告和承诺似的。所以它还在我的手腕上。在写代码时，我用余光瞟见它。它一直提醒我，我做了写出整洁代码的承诺。

---

[1] 原注：http://www.qualitytree.com/ 。

欢迎来到异步社区！

异步社区的来历

异步社区(www.epubit.com.cn)是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



异步社区
人民邮电出版社
技术圈
图书
电子书
文章
图书/作者/类别
写作
我们一步啦
异步社区成立一周年大型活动开启
我要写书
近期活动
+更多
异步社区成立一周年大型赠书活动开启！
异步社区的来历 异步社区是人民邮电出版社旗下IT 专，业图书旗舰社区，于2015 年8 月上线运营。异步社区依托于人民邮电出版社 20余 年的IT专业...
猫叔郭志敏 2016-08-02
阅读 575 推荐 2 收藏 0 评论 8
2016 iWeb峰会北京站即将开启，为HTML5娜喊！
每一次振臂高呼辐射行业的影响， 每一天无数人兢兢业业的勤奋，2016雄起！ 来吧，8月27日，HTML5峰会北京站，我在这里，等你来，为HTML5娜喊！...
猫叔郭志敏 2016-07-29
阅读 60 推荐 1 收藏 0 评论 0
周年庆满减促销 | 满100元减20元、满150元减35元、满200元减50元
+更多
CCIE路由和交换认证考试指南（第5版）（第1卷）
数据科学实战手册（R+Python）
软技能：代码之外的生存指南
Python密码学编程
Python游戏编程快速上手
机器学习项目开发实战
树莓派Python编程入门与实战（第2版）
像计算机科学家一样思考Python（第2版）
每周半价电子书
+更多
树莓派Python编程入门与实战（第2版）
[美] Richard Blum 勃鲁姆, Christine Bresnahan 布莱斯纳罕 (作者) 陈晓明 马立新 (译者)

社区里都有什么？

购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提

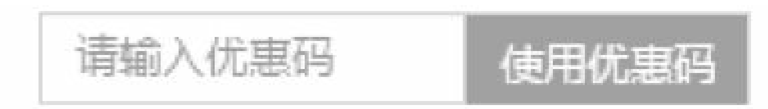
出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100积分=1元，购买图书时，在

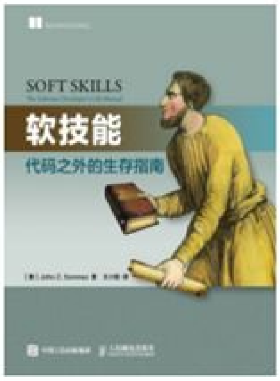


里填入可使用的积分数值，即可扣减相应金额。

> 特别优惠
>
> 购买本电子书的读者专享异步社区优惠券 。 使用方法：注册成为社区用户，在下单购书时输入“**57AWG** ”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。



软技能：代码之外的生存指南

[美]约翰 Z. 森梅兹（John Z. Sonmez）(作者)　王小刚 (译者)　杨海玲 (责任编辑)

分享　6 推荐　想读　9.0K 阅读

这是一本真正从“人”（而非技术也非管理）的角度关注软件开发人员自身发展的书。书中论述的内容既涉及生活习惯，又包括思维方式，凸显技术中“人”的因素，全面讲解软件行业从业人员所需知道的所有“软技能”。

本书聚焦于软件开发人员生活的方方面面，从揭秘面试的流程到精耕细作出一份杀手级简历，从创建大受欢迎的博客到打造你的个人品牌，从提高自己工作效率到与如何与“拖延症”做斗争，甚至包括如何投资不动产，如何关注自己的健康。

本书共分为职业篇、自我营销篇、学习篇、生产力篇、理财篇、健身篇、精神篇等七篇，概括了软件行业从业人员所需的“软技能”。

社区里还可以做什么？

提交勘误

　　您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

　　社区提供基于Markdown的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

　　如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

　　您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区

微信订阅号

微信服务号

官方微博

QQ群：368449889

社区网址： www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社-信息技术分社

投稿&咨询：contact@epubit.com.cn